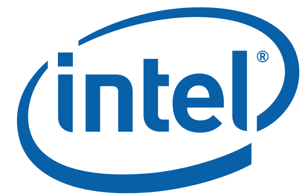


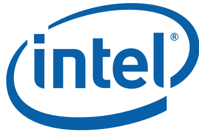
Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

March 2017

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

Document Number: 252046-054



Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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

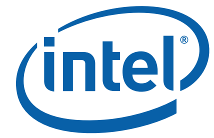
This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

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

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

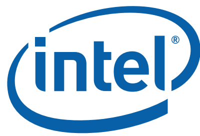
*Other names and brands may be claimed as the property of others.

Copyright © 1997-2017, Intel Corporation. All Rights Reserved.



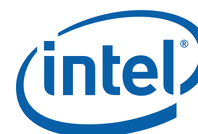
Contents

Revision History 4
Preface 7
Summary Tables of Changes 8
Documentation Changes. 9

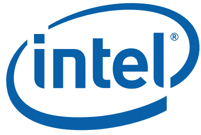


Revision History

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



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



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

§

Preface

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

Affected Documents

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

Nomenclature

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

Summary Tables of Changes

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

Codes Used in Summary Tables

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

Documentation Changes

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1, Volume 1
2	Updates to Chapter 8, Volume 1
3	Updates to Chapter 1, Volume 2
4	Updates to Chapter 3, Volume 2A
5	Updates to Chapter 4, Volume 2B
6	Updates to Chapter 5, Volume 2C
7	Updates to Chapter 1, Volume 3A
8	Updates to Chapter 2, Volume 3A
9	Updates to Chapter 4, Volume 3A
10	Updates to Chapter 5, Volume 3A
11	Updates to Chapter 17, Volume 3B
12	Updates to Chapter 18, Volume 3B
13	Updates to Chapter 19, Volume 3B
14	Updates to Chapter 22, Volume 3B
15	Updates to Chapter 25, Volume 3C
16	Updates to Chapter 30, Volume 3C
17	Updates to Chapter 35, Volume 3C
18	Updates to Chapter 40, Volume 3D
19	Updates to Chapter 1, Volume 4
20	Updates to Chapter 2, Volume 4

Documentation Changes

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

1. Updates to Chapter 1, Volume 1

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

Change to this chapter: Added reference to new volume 4: Model Specific Registers.

CHAPTER 1
ABOUT THIS MANUAL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual's content follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 NOTATIONAL CONVENTIONS

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

1.3.1 Bit and Byte Order

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

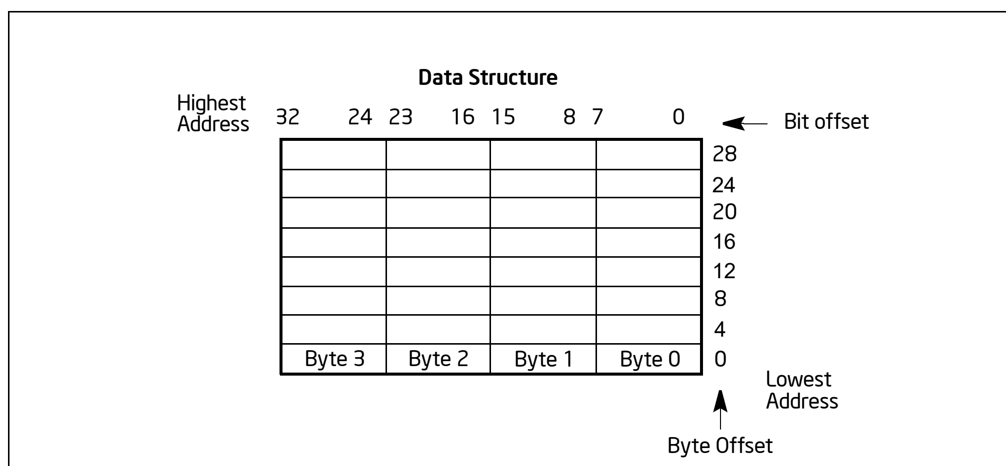


Figure 1-1. Bit and Byte Order

1.3.2 Reserved Bits and Software Compatibility

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

Software should follow these guidelines in dealing with reserved bits:

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

NOTE

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

1.3.2.1 Instruction Operands

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

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

where:

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

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

For example:

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

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

1.3.3 Hexadecimal and Binary Numbers

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

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

1.3.4 Segmented Addressing

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

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

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

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

```
DS:FF79H
```

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

CS:EIP

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

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

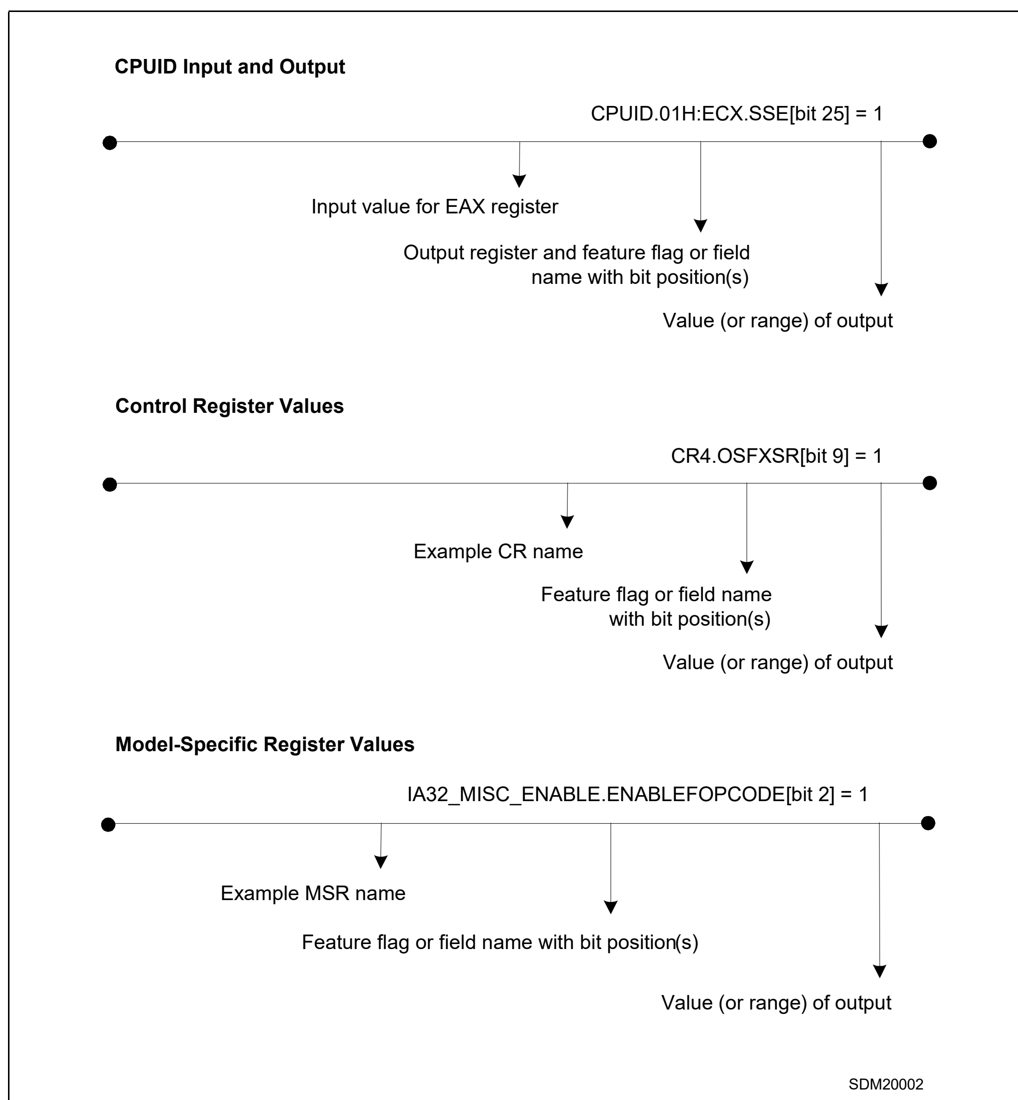


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.6 Exceptions

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

#PF(fault code)

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

#GP(0)

1.4 RELATED LITERATURE

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

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

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- Intel 64 Architecture x2APIC Specification:

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

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

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

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

More relevant links are:

- Intel® Developer Zone:

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

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

  http://www.intel.com/support/processors/
- Software products and packages:

  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® Hyper-Threading Technology (Intel® HT Technology):

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

2. Updates to Chapter 8, Volume 1

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

Change to this chapter: Corrected ES flag name in x87 FPU Status Register.

CHAPTER 8
PROGRAMMING WITH THE X87 FPU

The x87 Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities for use in graphics processing, scientific, engineering, and business applications. It supports the floating-point, integer, and packed BCD integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic.

This chapter describes the x87 FPU's execution environment and instruction set. It also provides exception handling information that is specific to the x87 FPU. Refer to the following chapters or sections of chapters for additional information about x87 FPU instructions and floating-point operations:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B,* provide detailed descriptions of x87 FPU instructions.
- Section 4.2.2, "Floating-Point Data Types," Section 4.2.1.2, "Signed Integers," and Section 4.7, "BCD and Packed BCD Integers," describe the floating-point, integer, and BCD data types.
- Section 4.9, "Overview of Floating-Point Exceptions," Section 4.9.1, "Floating-Point Exception Conditions," and Section 4.9.2, "Floating-Point Exception Priority," give an overview of the floating-point exceptions that the x87 FPU can detect and report.

8.1 X87 FPU EXECUTION ENVIRONMENT

The x87 FPU represents a separate execution environment within the IA-32 architecture (see Figure 8-1). This execution environment consists of eight data registers (called the x87 FPU data registers) and the following special-purpose registers:

- Status register
- Control register
- Tag word register
- Last instruction pointer register
- Last data (operand) pointer register
- Opcode register

These registers are described in the following sections.

The x87 FPU executes instructions from the processor's normal instruction stream. The state of the x87 FPU is independent from the state of the basic execution environment and from the state of SSE/SSE2/SSE3 extensions.

However, the x87 FPU and Intel MMX technology share state because the MMX registers are aliased to the x87 FPU data registers. Therefore, when writing code that uses x87 FPU and MMX instructions, the programmer must explicitly manage the x87 FPU and MMX state (see Section 9.5, "Compatibility with x87 FPU Architecture").

8.1.1 x87 FPU in 64-Bit Mode and Compatibility Mode

In compatibility mode and 64-bit mode, x87 FPU instructions function like they do in protected mode. Memory operands are specified using the ModR/M, SIB encoding that is described in Section 3.7.5, "Specifying an Offset."

8.1.2 x87 FPU Data Registers

The x87 FPU data registers (shown in Figure 8-1) consist of eight 80-bit registers. Values are stored in these registers in the double extended-precision floating-point format shown in Figure 4-3. When floating-point, integer, or packed BCD integer values are loaded from memory into any of the x87 FPU data registers, the values are automatically converted into double extended-precision floating-point format (if they are not already in that format). When computation results are subsequently transferred back into memory from any of the x87 FPU registers, the

results can be left in the double extended-precision floating-point format or converted back into a shorter floating-point format, an integer format, or the packed BCD integer format. (See Section 8.2, “x87 FPU Data Types,” for a description of the data types operated on by the x87 FPU.)

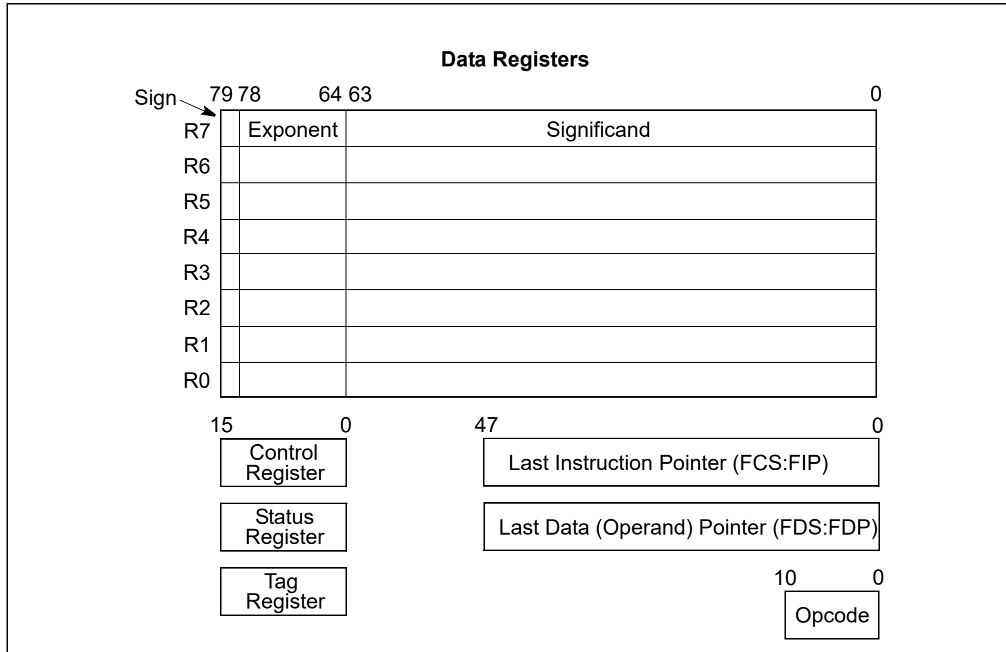


Figure 8-1. x87 FPU Execution Environment

The x87 FPU instructions treat the eight x87 FPU data registers as a register stack (see Figure 8-2). All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP (stack TOP) field in the x87 FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one. (For the x87 FPU, a load operation is equivalent to a push and a store operation is equivalent to a pop.) Note that load and store operations are also available that do not push and pop the stack.

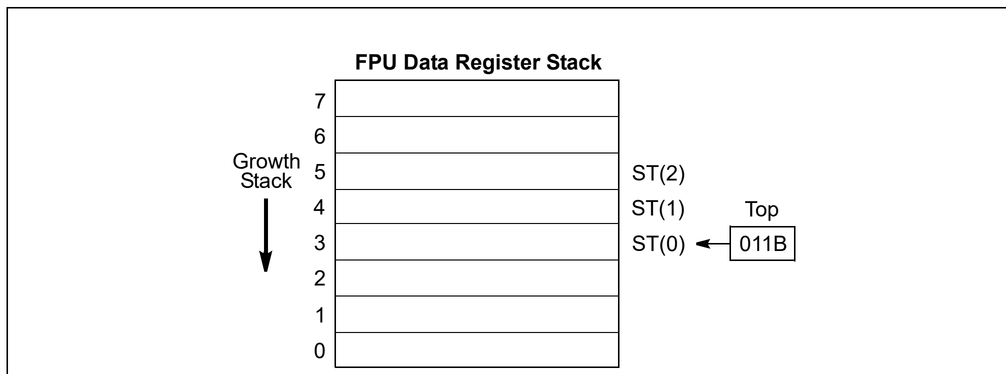


Figure 8-2. x87 FPU Data Register Stack

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be over-written (see Section 8.5.1.1, “Stack Overflow or Underflow Exception (#IS)”).

Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. Assemblers support these

register addressing modes, using the expression ST(0), or simply ST, to represent the current stack top and ST(i) to specify the *i*th register from TOP in the stack ($0 \leq i \leq 7$). For example, if TOP contains 011B (register 3 is the top of the stack), the following instruction would add the contents of two registers in the stack (registers 3 and 5):

```
FADD ST, ST(2);
```

Figure 8-3 shows an example of how the stack structure of the x87 FPU registers and instructions are typically used to perform a series of computations. Here, a two-dimensional dot product is computed, as follows:

1. The first instruction (FLD value1) decrements the stack register pointer (TOP) and loads the value 5.6 from memory into ST(0). The result of this operation is shown in snap-shot (a).
2. The second instruction multiplies the value in ST(0) by the value 2.4 from memory and stores the result in ST(0), shown in snap-shot (b).
3. The third instruction decrements TOP and loads the value 3.8 in ST(0).
4. The fourth instruction multiplies the value in ST(0) by the value 10.3 from memory and stores the result in ST(0), shown in snap-shot (c).
5. The fifth instruction adds the value and the value in ST(1) and stores the result in ST(0), shown in snap-shot (d).

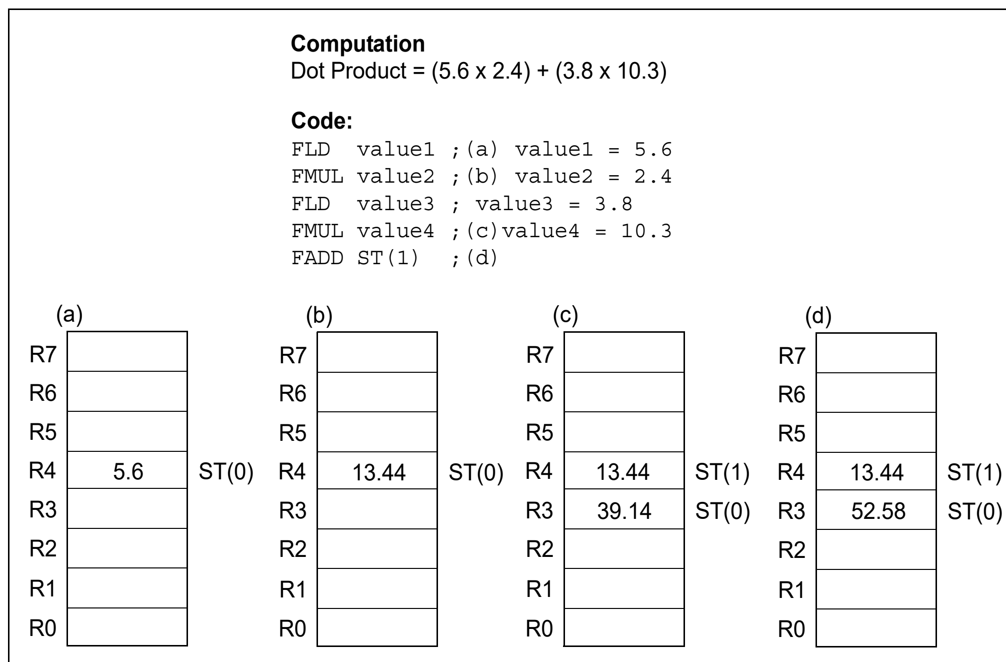


Figure 8-3. Example x87 FPU Dot Product Computation

The style of programming demonstrated in this example is supported by the floating-point instruction set. In cases where the stack structure causes computation bottlenecks, the FXCH (exchange x87 FPU register contents) instruction can be used to streamline a computation.

8.1.2.1 Parameter Passing With the x87 FPU Register Stack

Like the general-purpose registers, the contents of the x87 FPU data registers are unaffected by procedure calls, or in other words, the values are maintained across procedure boundaries. A calling procedure can thus use the x87 FPU data registers (as well as the procedure stack) for passing parameter between procedures. The called procedure can reference parameters passed through the register stack using the current stack register pointer (TOP) and the ST(0) and ST(i) nomenclature. It is also common practice for a called procedure to leave a return value or result in register ST(0) when returning execution to the calling procedure or program.

When mixing MMX and x87 FPU instructions in the procedures or code sequences, the programmer is responsible for maintaining the integrity of parameters being passed in the x87 FPU data registers. If an MMX instruction is executed before the parameters in the x87 FPU data registers have been passed to another procedure, the parameters may be lost (see Section 9.5, "Compatibility with x87 FPU Architecture").

8.1.3 x87 FPU Status Register

The 16-bit x87 FPU status register (see Figure 8-4) indicates the current state of the x87 FPU. The flags in the x87 FPU status register include the FPU busy flag, top-of-stack (TOP) pointer, condition code flags, exception summary status flag, stack fault flag, and exception flags. The x87 FPU sets the flags in this register to show the results of operations.

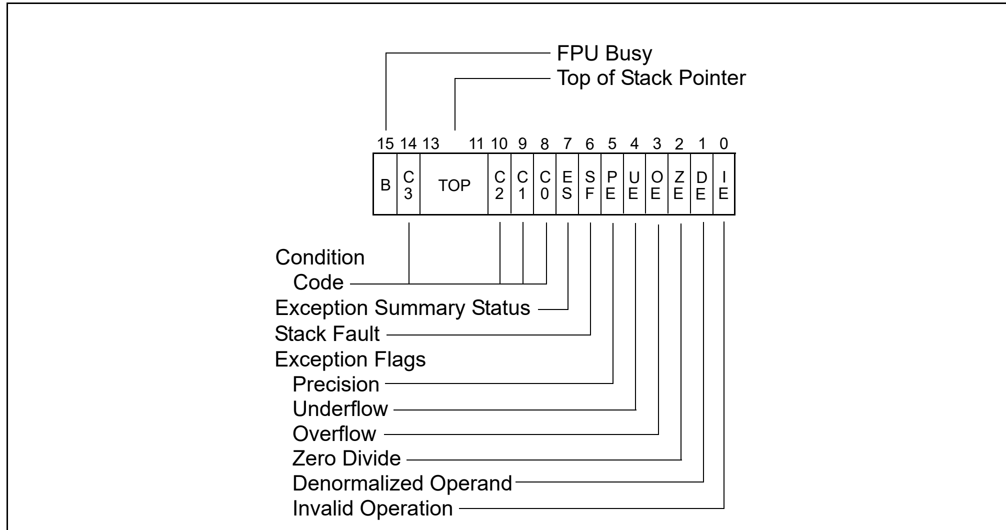


Figure 8-4. x87 FPU Status Word

The contents of the x87 FPU status register (referred to as the x87 FPU status word) can be stored in memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, FSAVE/FNSAVE, and FXSAVE instructions. It can also be stored in the AX register of the integer unit, using the FSTSW/FNSTSW instructions.

8.1.3.1 Top of Stack (TOP) Pointer

A pointer to the x87 FPU data register that is currently at the top of the x87 FPU register stack is contained in bits 11 through 13 of the x87 FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7. See Section 8.1.2, "x87 FPU Data Registers," for more information about the TOP pointer.

8.1.3.2 Condition Code Flags

The four condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. Table 8-1 summarizes the manner in which the floating-point instructions set the condition code flags. These condition code bits are used principally for conditional branching and for storage of information used in exception handling (see Section 8.1.4, "Branching and Conditional Moves on Condition Codes").

As shown in Table 8-1, the C1 condition code flag is used for a variety of functions. When both the IE and SF flags in the x87 FPU status word are set, indicating a stack overflow or underflow exception (#IS), the C1 flag distinguishes between overflow (C1 = 1) and underflow (C1 = 0). When the PE flag in the status word is set, indicating an inexact (rounded) result, the C1 flag is set to 1 if the last rounding by the instruction was upward. The FXAM instruction sets C1 to the sign of the value being examined.

The C2 condition code flag is used by the FPREM and FPREM1 instructions to indicate an incomplete reduction (or partial remainder). When a successful reduction has been completed, the C0, C3, and C1 condition code flags are set to the three least-significant bits of the quotient (Q2, Q1, and Q0, respectively). See “FPREM1—Partial Remainder” in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* for more information on how these instructions use the condition code flags.

The FPTAN, FSIN, FCOS, and FSINCOS instructions set the C2 flag to 1 to indicate that the source operand is beyond the allowable range of $\pm 2^{63}$ and clear the C2 flag if the source operand is within the allowable range.

Where the state of the condition code flags are listed as undefined in Table 8-1, do not rely on any specific value in these flags.

8.1.3.3 x87 FPU Floating-Point Exception Flags

The six x87 FPU floating-point exception flags (bits 0 through 5) of the x87 FPU status word indicate that one or more floating-point exceptions have been detected since the bits were last cleared. The individual exception flags (IE, DE, ZE, OE, UE, and PE) are described in detail in Section 8.4, “x87 FPU Floating-Point Exception Handling.” Each of the exception flags can be masked by an exception mask bit in the x87 FPU control word (see Section 8.1.5, “x87 FPU Control Word”). The exception summary status flag (ES, bit 7) is set when any of the unmasked exception flags are set. When the ES flag is set, the x87 FPU exception handler is invoked, using one of the techniques described in Section 8.7, “Handling x87 FPU Exceptions in Software.” (Note that if an exception flag is masked, the x87 FPU will still set the appropriate flag if the associated exception occurs, but it will not set the ES flag.)

The exception flags are “sticky” bits (once set, they remain set until explicitly cleared). They can be cleared by executing the FCLEX/FNCLEX (clear exceptions) instructions, by reinitializing the x87 FPU with the FINIT/FNINIT or FSAVE/FNSAVE instructions, or by overwriting the flags with an FRSTOR or FLDENV instruction.

The B-bit (bit 15) is included for 8087 compatibility only. It reflects the contents of the ES flag.

Table 8-1. Condition Code Interpretation

Instruction	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST, FUCOM, FUCOMP, FUCOMPP	Result of Comparison		Operands are not Comparable	0 or #IS
FCOMI, FCOMIP, FUCOMI, FUCOMIP	Undefined. (These instructions set the status flags in the EFLAGS register.)			#IS
FXAM	Operand class			Sign
FPREM, FPREM1	Q2	Q1	0 = reduction complete 1 = reduction incomplete	Q0 or #IS
F2XM1, FADD, FADDP, FBSTP, FCMOVcc, FIADD, FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FIMUL, FIST, FISTP, FISUB, FISUBR,FMUL, FMULP, FPATAN, FRNDINT, FSCALE, FST, FSTP, FSUB, FSUBP, FSUBR, FSUBRP,FSQRT, FYL2X, FYL2XP1	Undefined			Roundup or #IS
FCOS, FSIN, FSINCOS, FPTAN	Undefined		0 = source operand within range 1 = source operand out of range	Roundup or #IS (Undefined if C2 = 1)
FABS, FBLD, FCHS, FDECSTP, FILD, FINCSTP, FLD, Load Constants, FSTP (ext. prec.), FXCH, FXTRACT	Undefined			0 or #IS

Table 8-1. Condition Code Interpretation (Contd.)

FLDENV, FRSTOR	Each bit loaded from memory			
FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW,	Undefined			
FINIT/FNINIT, FSAVE/FNSAVE	0	0	0	0

8.1.3.4 Stack Fault Flag

The stack fault flag (bit 6 of the x87 FPU status word) indicates that stack overflow or stack underflow has occurred with data in the x87 FPU data register stack. The x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition.

When this flag is set, the condition code flag C1 indicates the nature of the fault: overflow (C1 = 1) and underflow (C1 = 0). The SF flag is a "sticky" flag, meaning that after it is set, the processor does not clear it until it is explicitly instructed to do so (for example, by an FINIT/FNINIT, FCLEX/FNCLEX, or FSAVE/FNSAVE instruction).

See Section 8.1.7, "x87 FPU Tag Word," for more information on x87 FPU stack faults.

8.1.4 Branching and Conditional Moves on Condition Codes

The x87 FPU (beginning with the P6 family processors) supports two mechanisms for branching and performing conditional moves according to comparisons of two floating-point values. These mechanism are referred to here as the "old mechanism" and the "new mechanism."

The old mechanism is available in x87 FPU's prior to the P6 family processors and in P6 family processors. This mechanism uses the floating-point compare instructions (FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, and FICOMP) to compare two floating-point values and set the condition code flags (C0 through C3) according to the results. The contents of the condition code flags are then copied into the status flags of the EFLAGS register using a two step process (see Figure 8-5):

1. The FSTSW AX instruction moves the x87 FPU status word into the AX register.
2. The SAHF instruction copies the upper 8 bits of the AX register, which includes the condition code flags, into the lower 8 bits of the EFLAGS register.

When the condition code flags have been loaded into the EFLAGS register, conditional jumps or conditional moves can be performed based on the new settings of the status flags in the EFLAGS register.

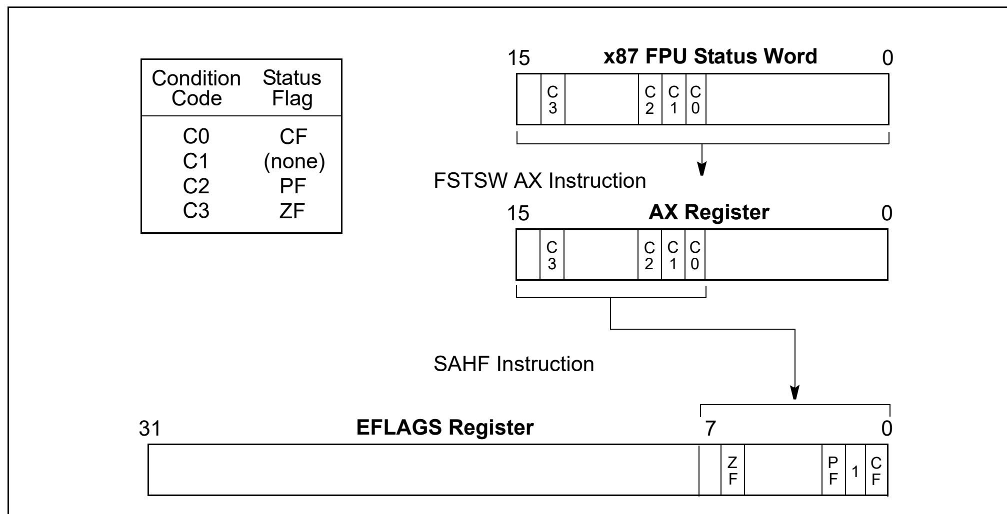


Figure 8-5. Moving the Condition Codes to the EFLAGS Register

The new mechanism is available beginning with the P6 family processors. Using this mechanism, the new floating-point compare and set EFLAGS instructions (FCOMI, FCOMIP, FUCOMI, and FUCOMIP) compare two floating-point values and set the ZF, PF, and CF flags in the EFLAGS register directly. A single instruction thus replaces the three instructions required by the old mechanism.

Note also that the FCMOV*cc* instructions (also new in the P6 family processors) allow conditional moves of floating-point values (values in the x87 FPU data registers) based on the setting of the status flags (ZF, PF, and CF) in the EFLAGS register. These instructions eliminate the need for an IF statement to perform conditional moves of floating-point values.

8.1.5 x87 FPU Control Word

The 16-bit x87 FPU control word (see Figure 8-6) controls the precision of the x87 FPU and rounding method used. It also contains the x87 FPU floating-point exception mask bits. The control word is cached in the x87 FPU control register. The contents of this register can be loaded with the FLDCW instruction and stored in memory with the FSTCW/FNSTCW instructions.

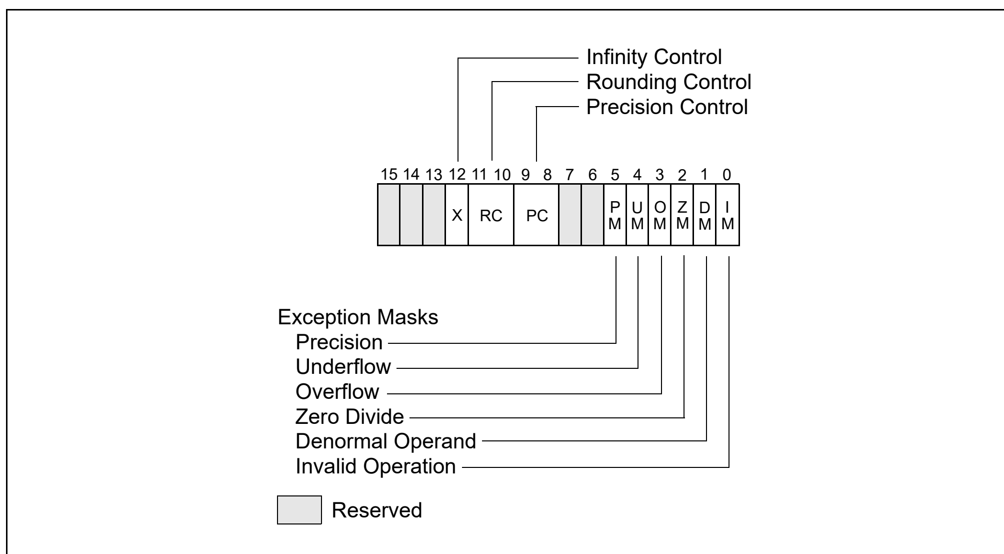


Figure 8-6. x87 FPU Control Word

When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU control word is set to 037FH, which masks all floating-point exceptions, sets rounding to nearest, and sets the x87 FPU precision to 64 bits.

8.1.5.1 x87 FPU Floating-Point Exception Mask Bits

The exception-flag mask bits (bits 0 through 5 of the x87 FPU control word) mask the 6 floating-point exception flags in the x87 FPU status word. When one of these mask bits is set, its corresponding x87 FPU floating-point exception is blocked from being generated.

8.1.5.2 Precision Control Field

The precision-control (PC) field (bits 8 and 9 of the x87 FPU control word) determines the precision (64, 53, or 24 bits) of floating-point calculations made by the x87 FPU (see Table 8-2). The default precision is double extended precision, which uses the full 64-bit significand available with the double extended-precision floating-point format of the x87 FPU data registers. This setting is best suited for most applications, because it allows applications to take full advantage of the maximum precision available with the x87 FPU data registers.

Table 8-2. Precision Control Field (PC)

Precision	PC Field
Single Precision (24 bits)	00B
Reserved	01B
Double Precision (53 bits)	10B
Double Extended Precision (64 bits)	11B

The double precision and single precision settings reduce the size of the significand to 53 bits and 24 bits, respectively. These settings are provided to support IEEE Standard 754 and to provide compatibility with the specifications of certain existing programming languages. Using these settings nullifies the advantages of the double extended-precision floating-point format's 64-bit significand length. When reduced precision is specified, the rounding of the significand value clears the unused bits on the right to zeros.

The precision-control bits only affect the results of the following floating-point instructions: FADD, FADDP, FIADD, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, and FSQRT.

8.1.5.3 Rounding Control Field

The rounding-control (RC) field of the x87 FPU control register (bits 10 and 11) controls how the results of x87 FPU floating-point instructions are rounded. See Section 4.8.4, "Rounding," for a discussion of rounding of floating-point values; See Section 4.8.4.1, "Rounding Control (RC) Fields", for the encodings of the RC field.

8.1.6 Infinity Control Flag

The infinity control flag (bit 12 of the x87 FPU control word) is provided for compatibility with the Intel 287 Math Coprocessor; it is not meaningful for later version x87 FPU coprocessors or IA-32 processors. See Section 4.8.3.3, "Signed Infinities," for information on how the x87 FPUs handle infinity values.

8.1.7 x87 FPU Tag Word

The 16-bit tag word (see Figure 8-7) indicates the contents of each the 8 registers in the x87 FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number (NaN, infinity, denormal, or unsupported format), or whether it is empty. The x87 FPU tag word is cached in the x87 FPU in the x87 FPU tag word register. When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU tag word is set to FFFFH, which marks all the x87 FPU data registers as empty.

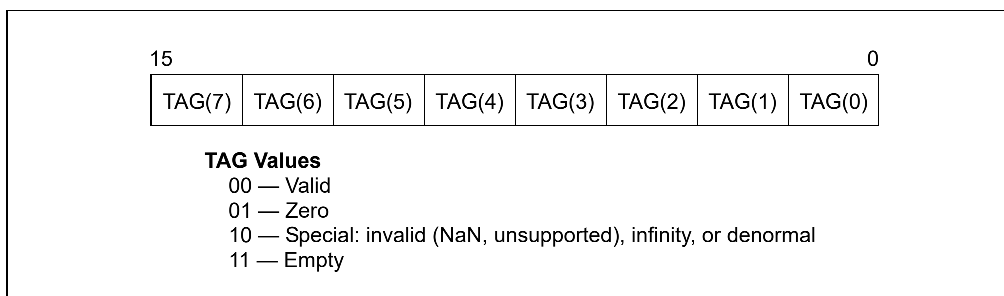


Figure 8-7. x87 FPU Tag Word

Each tag in the x87 FPU tag word corresponds to a physical register (numbers 0 through 7). The current top-of-stack (TOP) pointer stored in the x87 FPU status word can be used to associate tags with registers relative to ST(0).

The x87 FPU uses the tag values to detect stack overflow and underflow conditions (see Section 8.5.1.1, “Stack Overflow or Underflow Exception (#IS)”).

Application programs and exception handlers can use this tag information to check the contents of an x87 FPU data register without performing complex decoding of the actual data in the register. To read the tag register, it must be stored in memory using either the FSTENV/FNSTENV or FSAVE/FNSAVE instructions. The location of the tag word in memory after being saved with one of these instructions is shown in Figures 8-9 through 8-12.

Software cannot directly load or modify the tags in the tag register. The FLDENV and FRSTOR instructions load an image of the tag register into the x87 FPU; however, the x87 FPU uses those tag values only to determine if the data registers are empty (11B) or non-empty (00B, 01B, or 10B).

If the tag register image indicates that a data register is empty, the tag in the tag register for that data register is marked empty (11B); if the tag register image indicates that the data register is non-empty, the x87 FPU reads the actual value in the data register and sets the tag for the register accordingly. This action prevents a program from setting the values in the tag register to incorrectly represent the actual contents of non-empty data registers.

8.1.8 x87 FPU Instruction and Data (Operand) Pointers

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

Note that the value in the x87 FPU data pointer is always a pointer to a memory operand. If the last non-control instruction that was executed did not have a memory operand, the value in the data pointer is undefined (reserved). If CPUID.(EAX=07H,ECX=0H):EBX[bit 6] = 1, the data pointer is updated only for x87 non-control instructions that incur unmasked x87 exceptions.

The contents of the x87 FPU instruction and data pointers remain unchanged when any of the following instructions are executed: FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, and WAIT/FWAIT.

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

The x87 FPU instruction and data pointers each consists of an offset and a segment selector:

- The x87 FPU Instruction Pointer Offset (FIP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Instruction Pointer Selector (FCS) comprises 16 bits.
- The x87 FPU Data Pointer Offset (FDP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Data Pointer Selector (FDS) comprises 16 bits.

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

- FINIT/FNINIT. Each instruction clears FIP, FCS, FDP, and FDS.
- FLDENV, FRSTOR. These instructions use the memory formats given in Figures 8-9 through 8-12:
  - For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
  - If CR0.PE = 1, each instruction loads FCS and FDS from memory; otherwise, it clears them.
- FSAVE/FNSAVE, FSTENV/FNSTENV. These instructions use the memory formats given in Figures 8-9 through 8-12.
  - Each instruction saves the lower 32 bits of each FIP and FDP into memory. the upper 32 bits are not saved.
  - If CR0.PE = 1, each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.
  - After saving these data into memory, FSAVE/FNSAVE clears FIP, FCS, FDP, and FDS.

- FXRSTOR, XRSTOR. These instructions load data from a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.
  - Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
    - Each instruction loads FCS and FDS from memory.
  - In 64-bit mode with REX.W = 1, the instructions operate as follows:
    - Each instruction loads FIP and FDP from memory.
    - Each instruction clears FCS and FDS.
- FXSAVE, XSAVE, and XSAVEOPT. These instructions store data into a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.
  - Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - Each instruction saves the lower 32 bits of each of FIP and FDP into memory. The upper 32 bits are not saved.
    - Each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.
  - In 64-bit mode with REX.W = 1, each instruction saves FIP and FDP into memory. FCS and FDS are not saved.

8.1.9 Last Instruction Opcode

The x87 FPU stores in the 11-bit x87 FPU opcode register (FOP) the opcode of the last x87 non-control instruction executed that incurred an unmasked x87 exception. (This information provides state information for exception handlers.) Only the first and second opcode bytes (after all prefixes) are stored in the x87 FPU opcode register. Figure 8-8 shows the encoding of these two bytes. Since the upper 5 bits of the first opcode byte are the same for all floating-point opcodes (11011B), only the lower 3 bits of this byte are stored in the opcode register.

8.1.9.1 Fopcode Compatibility Sub-mode

Some Pentium 4 and Intel Xeon processors provide program control over the value stored into FOP. Here, bit 2 of the IA32_MISC_ENABLE MSR enables (set) or disables (clear) the fopcode compatibility mode.

If fopcode compatibility mode is enabled, FOP is defined as it had been in previous IA-32 implementations, as the opcode of the last x87 non-control instruction executed (even if that instruction did not incur an unmasked x87 exception).

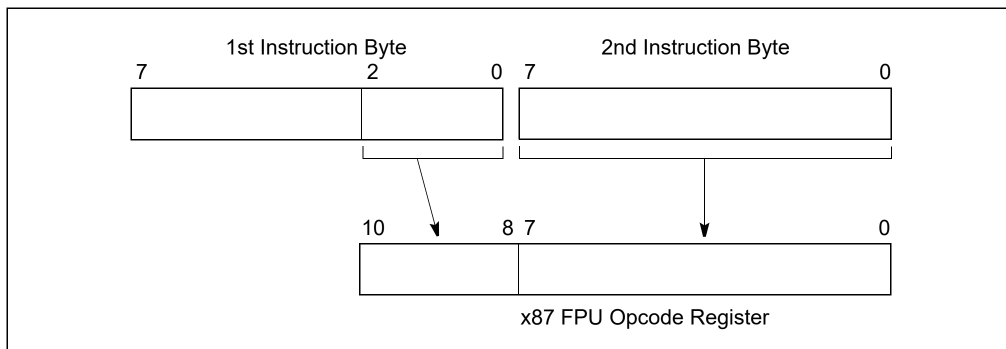


Figure 8-8. Contents of x87 FPU Opcode Registers

The fopcode compatibility mode should be enabled only when x87 FPU floating-point exception handlers are designed to use the fopcode to analyze program performance or restart a program after an exception has been handled.

More recent Intel 64 processors do not support fopcode compatibility mode and do not allow software to set bit 2 of the IA32_MISC_ENABLE MSR.

8.1.10 Saving the x87 FPU's State with FSTENV/FNSTENV and FSAVE/FNSAVE

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions store x87 FPU state information in memory for use by exception handlers and other system and application software. The FSTENV/FNSTENV instruction saves the contents of the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers. The FSAVE/FNSAVE instruction stores that information plus the contents of the x87 FPU data registers. Note that the FSAVE/FNSAVE instruction also initializes the x87 FPU to default values (just as the FINIT/FNINIT instruction does) after it has saved the original state of the x87 FPU.

The manner in which this information is stored in memory depends on the operating mode of the processor (protected mode or real-address mode) and on the operand-size attribute in effect (32-bit or 16-bit). See Figures 8-9 through 8-12. In virtual-8086 mode or SMM, the real-address mode formats shown in Figure 8-12 is used. See Chapter 34, "System Management Mode," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C,* for information on using the x87 FPU while in SMM.

The FLDENV and FRSTOR instructions allow x87 FPU state information to be loaded from memory into the x87 FPU. Here, the FLDENV instruction loads only the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers, and the FRSTOR instruction loads all the x87 FPU registers, including the x87 FPU stack registers.

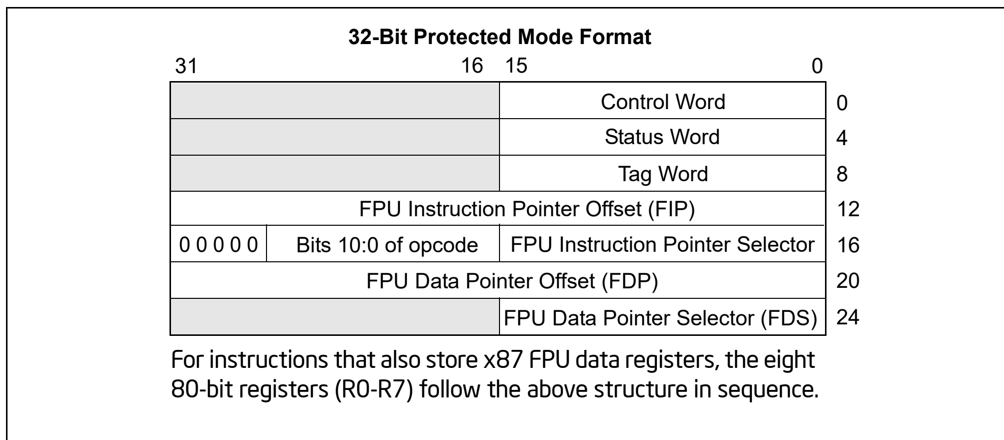


Figure 8-9. Protected Mode x87 FPU State Image in Memory, 32-Bit Format

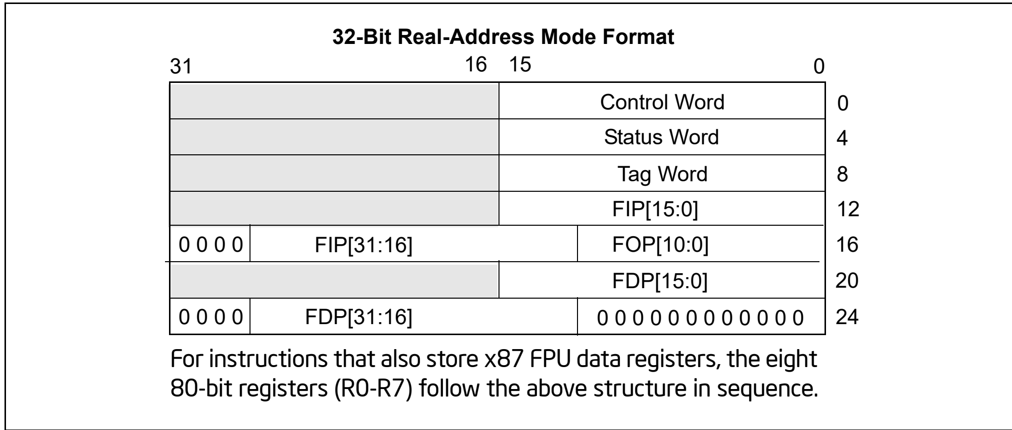


Figure 8-10. Real Mode x87 FPU State Image in Memory, 32-Bit Format

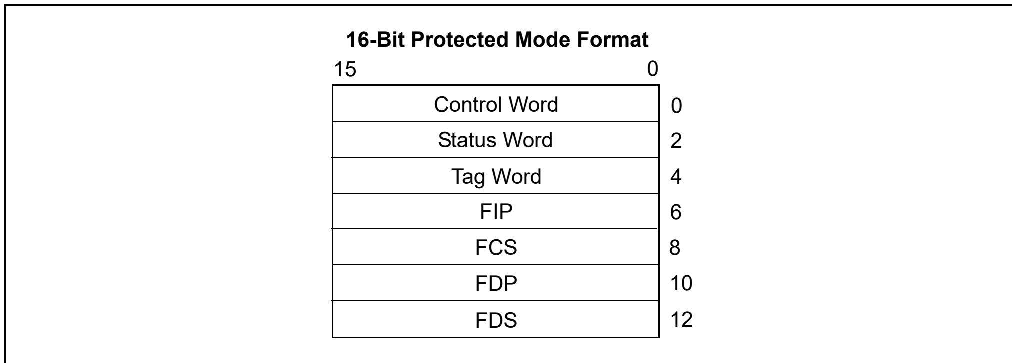


Figure 8-11. Protected Mode x87 FPU State Image in Memory, 16-Bit Format

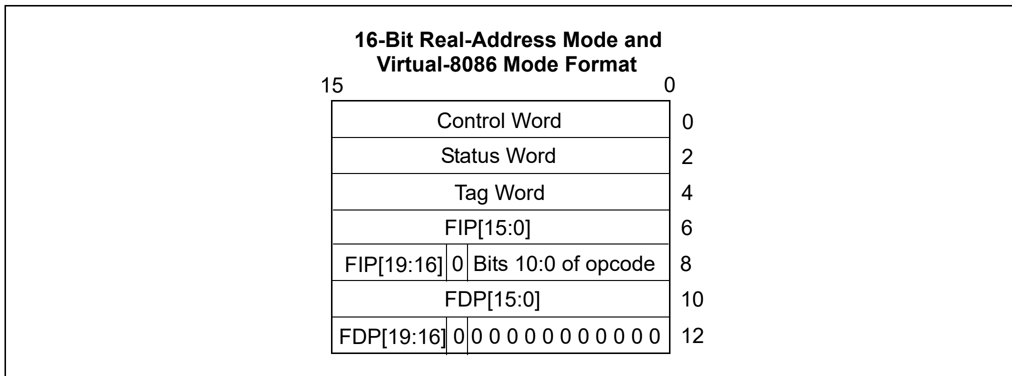


Figure 8-12. Real Mode x87 FPU State Image in Memory, 16-Bit Format

8.1.11 Saving the x87 FPU's State with FXSAVE

The FXSAVE and FXRSTOR instructions save and restore, respectively, the x87 FPU state along with the state of the XMM registers and the MXCSR register. Using the FXSAVE instruction to save the x87 FPU state has two benefits: (1) FXSAVE executes faster than FSAVE, and (2) FXSAVE saves the entire x87 FPU, MMX, and XMM state in one operation. See Section 10.5, "FXSAVE and FXRSTOR Instructions," for additional information about these instructions.

8.2 X87 FPU DATA TYPES

The x87 FPU recognizes and operates on the following seven data types (see Figures 8-13): single-precision floating point, double-precision floating point, double extended-precision floating point, signed word integer, signed doubleword integer, signed quadword integer, and packed BCD decimal integers.

For detailed information about these data types, see Section 4.2.2, "Floating-Point Data Types," Section 4.2.1.2, "Signed Integers," and Section 4.7, "BCD and Packed BCD Integers."

With the exception of the 80-bit double extended-precision floating-point format, all of these data types exist in memory only. When they are loaded into x87 FPU data registers, they are converted into double extended-precision floating-point format and operated on in that format.

Denormal values are also supported in each of the floating-point types, as required by IEEE Standard 754. When a denormal number in single-precision or double-precision floating-point format is used as a source operand and the denormal exception is masked, the x87 FPU automatically normalizes the number when it is converted to double extended-precision format.

When stored in memory, the least significant byte of an x87 FPU data-type value is stored at the initial address specified for the value. Successive bytes from the value are then stored in successively higher addresses in memory. The floating-point instructions load and store memory operands using only the initial address of the operand.

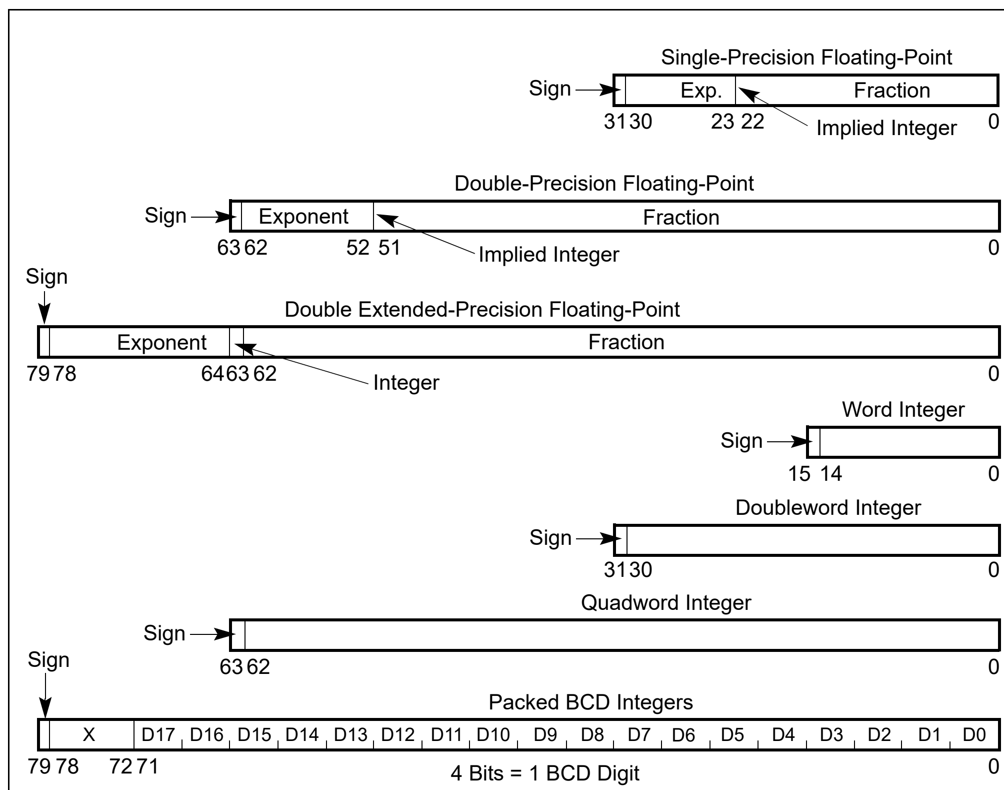


Figure 8-13. x87 FPU Data Type Formats

As a general rule, values should be stored in memory in double-precision format. This format provides sufficient range and precision to return correct results with a minimum of programmer attention. The single-precision format is useful for debugging algorithms, because rounding problems will manifest themselves more quickly in this format. The double extended-precision format is normally reserved for holding intermediate results in the x87 FPU registers and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. However, when an application requires the maximum range and precision of the x87 FPU (for data storage, computations, and results), values can be stored in memory in double extended-precision format.

8.2.1 Indefinites

For each x87 FPU data type, one unique encoding is reserved for representing the special value **indefinite**. The x87 FPU produces indefinite values as responses to some masked floating-point invalid-operation exceptions. See Tables 4-1, 4-3, and 4-4 for the encoding of the integer indefinite, QNaN floating-point indefinite, and packed BCD integer indefinite, respectively.

The binary integer encoding 100..00B represents either of two things, depending on the circumstances of its use:

- The largest negative number supported by the format (-2^{15}, -2^{31}, or -2^{63})
- The **integer indefinite** value

If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the x87 FPU interprets it as the largest negative number representable in the format being used. If the x87 FPU detects an invalid operation when storing an integer value in memory with an FIST/FISTP instruction and the invalid-operation exception is masked, the x87 FPU stores the integer indefinite encoding in the destination operand as a masked response to the exception. In situations where the origin of a value with this encoding may be ambiguous, the invalid-operation exception flag can be examined to see if the value was produced as a response to an exception.

8.2.2 Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals

The double extended-precision floating-point format permits many encodings that do not fall into any of the categories shown in Table 4-3. Table 8-3 shows these unsupported encodings. Some of these encodings were supported by the Intel 287 math coprocessor; however, most of them are not supported by the Intel 387 math coprocessor and later IA-32 processors. These encodings are no longer supported due to changes made in the final version of IEEE Standard 754 that eliminated these encodings.

Specifically, the categories of encodings formerly known as pseudo-NaNs, pseudo-infinities, and un-normal numbers are not supported and should not be used as operand values. The Intel 387 math coprocessor and later IA-32 processors generate an invalid-operation exception when these encodings are encountered as operands.

Beginning with the Intel 387 math coprocessor, the encodings formerly known as pseudo-denormal numbers are not generated by IA-32 processors. When encountered as operands, however, they are handled correctly; that is, they are treated as denormals and a denormal exception is generated. Pseudo-denormal numbers should not be used as operand values. They are supported by current IA-32 processors (as described here) to support legacy code.

Table 8-3. Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals

Class		Sign	Biased Exponent	Significand	
				Integer	Fraction
Positive Pseudo-NaNs	Quiet	0 . 0	11..11 . 11..11	0	11..11 . 10..00
	Signaling	0 . 0	11..11 . 11..11	0	01..11 . 00..01
Positive Floating Point	Pseudo-infinity	0	11..11	0	00..00
	Unnormals	0 . 0	11..10 . 00..01	0	11..11 . 00..00
	Pseudo-denormals	0 . 0	00..00 . 00..00	1	11..11 . 00..00

Table 8-3. Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals (Contd.)

Negative Floating Point	Pseudo-denormals	1 . 1	00..00 . 00..00	1	11..11 . 00..00
	Unnormals	1 . 1	11..10 . 00..01	0	11..01 . 00..00
	Pseudo-infinity	1	11..11	0	00..00
Negative Pseudo-NaNs	Signaling	1 . 1	11..11 . 11..11	0	01..11 . 00..01
	Quiet	1 . 1	11..11 . 11..11	0	11..11 . 10..00
			← 15 bits →		← 63 bits →

8.3 X87 FPU INSTRUCTION SET

The floating-point instructions that the x87 FPU supports can be grouped into six functional categories:

- Data transfer instructions
- Basic arithmetic instructions
- Comparison instructions
- Transcendental instructions
- Load constant instructions
- x87 FPU control instructions

See Section , "CPUID.EAX=80000001H:ECX.PREFTEHCHW[bit 8]: if 1 indicates the processor supports the PREFTEHCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFTEHCHWT1[bit 0]: if 1 indicates the processor supports the PREFTEHCHWT1 instruction.," for a list of the floating-point instructions by category.

The following section briefly describes the instructions in each category. Detailed descriptions of the floating-point instructions are given in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

8.3.1 Escape (ESC) Instructions

All of the instructions in the x87 FPU instruction set fall into a class of instructions known as escape (ESC) instructions. All of these instructions have a common opcode format, where the first byte of the opcode is one of the numbers from D8H through DFH.

8.3.2 x87 FPU Instruction Operands

Most floating-point instructions require one or two operands, located on the x87 FPU data-register stack or in memory. (None of the floating-point instructions accept immediate operands.)

When an operand is located in a data register, it is referenced relative to the ST(0) register (the register at the top of the register stack), rather than by a physical register number. Often the ST(0) register is an implied operand.

Operands in memory can be referenced using the same operand addressing methods described in Section 3.7, "Operand Addressing."

8.3.3 Data Transfer Instructions

The data transfer instructions (see Table 8-4) perform the following operations:

- Load a floating-point, integer, or packed BCD operand from memory into the ST(0) register.
- Store the value in an ST(0) register to memory in floating-point, integer, or packed BCD format.
- Move values between registers in the x87 FPU register stack.

The FLD (load floating point) instruction pushes a floating-point operand from memory onto the top of the x87 FPU data-register stack. If the operand is in single-precision or double-precision floating-point format, it is automatically converted to double extended-precision floating-point format. This instruction can also be used to push the value in a selected x87 FPU data register onto the top of the register stack.

The FILD (load integer) instruction converts an integer operand in memory into double extended-precision floating-point format and pushes the value onto the top of the register stack. The FBLD (load packed decimal) instruction performs the same load operation for a packed BCD operand in memory.

Table 8-4. Data Transfer Instructions

Floating Point		Integer		Packed Decimal	
FLD	Load Floating Point	FILD	Load Integer	FBLD	Load Packed Decimal
FST	Store Floating Point	FIST	Store Integer		
FSTP	Store Floating Point and Pop	FISTP	Store Integer and Pop	FBSTP	Store Packed Decimal and Pop
FXCH	Exchange Register Contents				
FCMOV*cc*	Conditional Move				

The FST (store floating point) and FIST (store integer) instructions store the value in register ST(0) in memory in the destination format (floating point or integer, respectively). Again, the format conversion is carried out automatically.

The FSTP (store floating point and pop), FISTP (store integer and pop), and FBSTP (store packed decimal and pop) instructions store the value in the ST(0) registers into memory in the destination format (floating point, integer, or packed BCD), then performs a **pop** operation on the register stack. A pop operation causes the ST(0) register to be marked empty and the stack pointer (TOP) in the x87 FPU control work to be incremented by 1. The FSTP instruction can also be used to copy the value in the ST(0) register to another x87 FPU register [ST(i)].

The FXCH (exchange register contents) instruction exchanges the value in a selected register in the stack [ST(i)] with the value in ST(0).

The FCMOV*cc* (conditional move) instructions move the value in a selected register in the stack [ST(i)] to register ST(0) if a condition specified with a condition code (*cc*) is satisfied (see Table 8-5). The condition being tested for is represented by the status flags in the EFLAGS register. The condition code mnemonics are appended to the letters "FCMOV" to form the mnemonic for a FCMOV*cc* instruction.

Table 8-5. Floating-Point Conditional Move Instructions

Instruction Mnemonic	Status Flag States	Condition Description
FCMOVB	CF=1	Below
FCMOVNB	CF=0	Not below
FCMOVE	ZF=1	Equal
FCMOVNE	ZF=0	Not equal
Instruction Mnemonic	**Status Flag States**	**Condition Description**
FCMOVBE	CF=1 or ZF=1	Below or equal
FCMOVNBE	CF=0 or ZF=0	Not below nor equal
FCMOVU	PF=1	Unordered
FCMOVNU	PF=0	Not unordered

Like the CMOV*cc* instructions, the FCMOV*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction.

8.3.4 Load Constant Instructions

The following instructions push commonly used constants onto the top [ST(0)] of the x87 FPU register stack:

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

The constant values have full double extended-precision floating-point precision (64 bits) and are accurate to approximately 19 decimal digits. They are stored internally in a format more precise than double extended-precision floating point. When loading the constant, the x87 FPU rounds the more precise internal constant according to the RC (rounding control) field of the x87 FPU control word. The inexact-result exception (#P) is not generated as a result of this rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up. See Section 8.3.8, “Approximation of Pi,” for information on the π constant.

8.3.5 Basic Arithmetic Instructions

The following floating-point instructions perform basic arithmetic operations on floating-point numbers. Where applicable, these instructions match IEEE Standard 754:

FADD/FADDP	Add floating point
FIADD	Add integer to floating point
FSUB/FSUBP	Subtract floating point
FISUB	Subtract integer from floating point
FSUBR/FSUBRP	Reverse subtract floating point
FISUBR	Reverse subtract floating point from integer
FMUL/FMULP	Multiply floating point
FIMUL	Multiply integer by floating point
FDIV/FDIVP	Divide floating point
FIDIV	Divide floating point by integer
FDIVR/FDIVRP	Reverse divide
FIDIVR	Reverse divide integer by floating point
FABS	Absolute value
FCHS	Change sign
FSQRT	Square root
FPREM	Partial remainder
FPREM1	IEEE partial remainder
FRNDINT	Round to integral value
FXTRACT	Extract exponent and significand

The add, subtract, multiply and divide instructions operate on the following types of operands:

- Two x87 FPU data registers
- An x87 FPU data register and a floating-point or integer value in memory

See Section 8.1.2, “x87 FPU Data Registers,” for a description of how operands are referenced on the data register stack.

Operands in memory can be in single-precision floating-point, double-precision floating-point, word-integer, or doubleword-integer format. They are converted to double extended-precision floating-point format automatically.

Reverse versions of the subtract (FSUBR) and divide (FDIVR) instructions enable efficient coding. For example, the following options are available with the FSUB and FSUBR instructions for operating on values in a specified x87 FPU data register ST(*i*) and the ST(0) register:

FSUB:

ST(0) ← ST(0) – ST(*i*)
ST(*i*) ← ST(*i*) – ST(0)

FSUBR:

ST(0) ← ST(*i*) – ST(0)
ST(*i*) ← ST(0) – ST(*i*)

These instructions eliminate the need to exchange values between the ST(0) register and another x87 FPU register to perform a subtraction or division.

The pop versions of the add, subtract, multiply, and divide instructions offer the option of popping the x87 FPU register stack following the arithmetic operation. These instructions operate on values in the ST(*i*) and ST(0) registers, store the result in the ST(*i*) register, and pop the ST(0) register.

The FPREM instruction computes the remainder from the division of two operands in the manner used by the Intel 8087 and Intel 287 math coprocessors; the FPREM1 instruction computes the remainder in the manner specified in IEEE Standard 754.

The FSQRT instruction computes the square root of the source operand.

The FRNDINT instruction returns a floating-point value that is the integral value closest to the source value in the direction of the rounding mode specified in the RC field of the x87 FPU control word.

The FABS, FCHS, and FXTRACT instructions perform convenient arithmetic operations. The FABS instruction produces the absolute value of the source operand. The FCHS instruction changes the sign of the source operand. The FXTRACT instruction separates the source operand into its exponent and fraction and stores each value in a register in floating-point format.

8.3.6 Comparison and Classification Instructions

The following instructions compare or classify floating-point values:

FCOM/FCOMP/FCOMPPCompare floating point and set x87 FPU condition code flags.

FUCOM/FUCOMP/FUCOMPPUnordered compare floating point and set x87 FPU condition code flags.

FICOM/FICOMPCompare integer and set x87 FPU condition code flags.

FCOMI/FCOMIPCompare floating point and set EFLAGS status flags.

FUCOMI/FUCOMIPUnordered compare floating point and set EFLAGS status flags.

FTST Test (compare floating point with 0.0). FXAMExamine.

Comparison of floating-point values differ from comparison of integers because floating-point values have four (rather than three) mutually exclusive relationships: less than, equal, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN or in an unsupported format. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have less than, equal, or greater than relationships with other floating-point values.

The FCOM, FCOMP, and FCOMPP instructions compare the value in register ST(0) with a floating-point source operand and set the condition code flags (C0, C2, and C3) in the x87 FPU status word according to the results (see Table 8-6).

If an unordered condition is detected (one or both of the values are NaNs or in an undefined format), a floating-point invalid-operation exception is generated.

The pop versions of the instruction pop the x87 FPU register stack once or twice after the comparison operation is complete.

The FUCOM, FUCOMP, and FUCOMPP instructions operate the same as the FCOM, FCOMP, and FCOMPP instructions. The only difference is that with the FUCOM, FUCOMP, and FUCOMPP instructions, if an unordered condition is detected because one or both of the operands are QNaNs, the floating-point invalid-operation exception is not generated.

Table 8-6. Setting of x87 FPU Condition Code Flags for Floating-Point Number Comparisons

Condition	C3	C2	C0
ST(0) > Source Operand	0	0	0
ST(0) < Source Operand	0	0	1
ST(0) = Source Operand	1	0	0
Unordered	1	1	1

The FICOM and FICOMP instructions also operate the same as the FCOM and FCOMP instructions, except that the source operand is an integer value in memory. The integer value is automatically converted into an double extended-precision floating-point value prior to making the comparison. The FICOMP instruction pops the x87 FPU register stack following the comparison operation.

The FTST instruction performs the same operation as the FCOM instruction, except that the value in register ST(0) is always compared with the value 0.0.

The FCOMI and FCOMIP instructions were introduced into the IA-32 architecture in the P6 family processors. They perform the same comparison as the FCOM and FCOMP instructions, except that they set the status flags (ZF, PF, and CF) in the EFLAGS register to indicate the results of the comparison (see Table 8-7) instead of the x87 FPU condition code flags. The FCOMI and FCOMIP instructions allow condition branch instructions (J*cc*) to be executed directly from the results of their comparison.

Table 8-7. Setting of EFLAGS Status Flags for Floating-Point Number Comparisons

Comparison Results	ZF	PF	CF
ST0 > ST(*i*)	0	0	0
ST0 < ST(*i*)	0	0	1
ST0 = ST(*i*)	1	0	0
Unordered	1	1	1

Software can check if the FCOMI and FCOMIP instructions are supported by checking the processor's feature information with the CPUID instruction.

The FUCOMI and FUCOMIP instructions operate the same as the FCOMI and FCOMIP instructions, except that they do not generate a floating-point invalid-operation exception if the unordered condition is the result of one or both of the operands being a QNaN. The FCOMIP and FUCOMIP instructions pop the x87 FPU register stack following the comparison operation.

The FXAM instruction determines the classification of the floating-point value in the ST(0) register (that is, whether the value is zero, a denormal number, a normal finite number, ∞, a NaN, or an unsupported format) or that the register is empty. It sets the x87 FPU condition code flags to indicate the classification (see "FXAM—Examine" in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A)*. It also sets the C1 flag to indicate the sign of the value.

8.3.6.1 Branching on the x87 FPU Condition Codes

The processor does not offer any control-flow instructions that branch on the setting of the condition code flags (C0, C2, and C3) in the x87 FPU status word. To branch on the state of these flags, the x87 FPU status word must

first be moved to the AX register in the integer unit. The FSTSW AX (store status word) instruction can be used for this purpose. When these flags are in the AX register, the TEST instruction can be used to control conditional branching as follows:

1. Check for an unordered result. Use the TEST instruction to compare the contents of the AX register with the constant 0400H (see Table 8-8). This operation will clear the ZF flag in the EFLAGS register if the condition code flags indicate an unordered result; otherwise, the ZF flag will be set. The JNZ instruction can then be used to transfer control (if necessary) to a procedure for handling unordered operands.

Table 8-8. TEST Instruction Constants for Conditional Branching

Order	Constant	Branch
ST(0) > Source Operand	4500H	JZ
ST(0) < Source Operand	0100H	JNZ
ST(0) = Source Operand	4000H	JNZ
Unordered	0400H	JNZ

2. Check ordered comparison result. Use the constants given in Table 8-8 in the TEST instruction to test for a less than, equal to, or greater than result, then use the corresponding conditional branch instruction to transfer program control to the appropriate procedure or section of code.

If a program or procedure has been thoroughly tested and it incorporates periodic checks for QNaN results, then it is not necessary to check for the unordered result every time a comparison is made.

See Section 8.1.4, "Branching and Conditional Moves on Condition Codes," for another technique for branching on x87 FPU condition codes.

Some non-comparison x87 FPU instructions update the condition code flags in the x87 FPU status word. To ensure that the status word is not altered inadvertently, store it immediately following a comparison operation.

8.3.7 Trigonometric Instructions

The following instructions perform four common trigonometric functions:

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Tangent
FPATAN	Arctangent

These instructions operate on the top one or two registers of the x87 FPU register stack and they return their results to the stack. The source operands for the FSIN, FCOS, FSINCOS, and FPTAN instructions must be given in radians; the source operand for the FPATAN instruction is given in rectangular coordinate units.

The FSINCOS instruction returns both the sine and the cosine of a source operand value. It operates faster than executing the FSIN and FCOS instructions in succession.

The FPATAN instruction computes the arctangent of ST(1) divided by ST(0), returning a result in radians. It is useful for converting rectangular coordinates to polar coordinates.

See Section 8.3.8, "Approximation of Pi" and Section 8.3.10, "Transcendental Instruction Accuracy" for information regarding the accuracy of these instructions.

8.3.8 Approximation of Pi

When the argument (source operand) of a trigonometric function is within the domain of the function, the argument is automatically reduced by the appropriate multiple of 2π through the same reduction mechanism used by the FPREM and FPREM1 instructions. The internal value of π (3.1415926…) that the x87 FPU uses for argument

reduction and other computations, denoted as Pi in the expression below. The numerical value of Pi can be written as:

$Pi = 0.f * 2^2$

where the fraction f is expressed in binary form as:

f = C90FDAA2 2168C234 C

(The spaces in the fraction above indicate 32-bit boundaries.)

The internal approximation Pi of the value π has a 66 significant bits. Since the exact value of π represented in binary has the next 3 bits equal to 0, it means that Pi is the value of π rounded to nearest-even to 68 bits, and also the value of π rounded toward zero (truncated) to 69 bits.

However, accuracy problems may arise because this relatively short finite approximation Pi of the number π is used for calculating the reduced argument of the trigonometric function approximations in the implementations of FSIN, FCOS, FSINCOS, and FPTAN. Alternately, this means that FSIN (x), FCOS (x), and FPTAN (x) are really approximating the mathematical functions sin (x * π /Pi), cos (x * π / Pi), and tan (x * π / Pi), and not exactly sin (x), cos (x), and tan (x). (Note that FSINCOS is the equivalent of FSIN and FCOS combined together). The period of sin (x * π /Pi) for example is 2* Pi, and not 2π.

See also Section 8.3.10, “Transcendental Instruction Accuracy” for more information on the accuracy of these functions.

8.3.9 Logarithmic, Exponential, and Scale

The following instructions provide two different logarithmic functions, an exponential function and a scale function:

FYL2X	Logarithm
FYL2XP1	Logarithm epsilon
F2XM1	Exponential
FSCALE	Scale

The FYL2X and FYL2XP1 instructions perform two different base 2 logarithmic operations. The FYL2X instruction computes ($y * \log_2 x$). This operation permits the calculation of the log of any base using the following equation:

$\log_b x = (1/\log_2 b) * \log_2 x$

The FYL2XP1 instruction computes ($y * \log_2(x + 1)$). This operation provides optimum accuracy for values of x that are close to 0.

The F2XM1 instruction computes ($2^x - 1$). This instruction only operates on source values in the range –1.0 to +1.0.

The FSCALE instruction multiplies the source operand by a power of 2.

8.3.10 Transcendental Instruction Accuracy

New transcendental instruction algorithms were incorporated into the IA-32 architecture beginning with the Pentium processors. These new algorithms (used in transcendental instructions FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) allow a higher level of accuracy than was possible in earlier IA-32 processors and x87 math coprocessors. The accuracy of these instructions is measured in terms of **units in the last place** (**ulp**). For a given argument *x*, let *f*(*x*) and *F*(*x*) be the correct and computed (approximate) function values, respectively. The error in ulps is defined to be:

$$error = \left|\frac{f(x) - F(x)}{2^{k-63}}\right|$$

where *k* is an integer such that:

$$1 \le 2^{-k} f(x) < 2.$$

With the Pentium processor and later IA-32 processors, the worst case error on transcendental functions is less than 1 ulp when rounding to the nearest (even) and less than 1.5 ulps when rounding in other modes. The functions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

However, for FSIN, FCOS, FSINCOS, and FPTAN which approximate periodic trigonometric functions, the previous statement about maximum ulp errors is true only when these instructions are applied to reduced argument (see Section 8.3.8, "Approximation of Pi"). This is due to the fact that only 66 significant bits are retained in the finite approximation Pi of the number π (3.1415926…), used internally for calculating the reduced argument in FSIN, FCOS, FSINCOS, and FPTAN. This approximation of π is not always sufficiently accurate for good argument reduction.

For single precision, the argument of FSIN, FCOS, FSINCOS, and FPTAN must exceed 200,000 radians in order for the error of the result to exceed 1 ulp when rounding to the nearest (even), or 1.5 ulps when rounding in other (directed) rounding modes.

For double and double-extended precision, the ulp errors will grow above these thresholds for arguments much smaller in magnitude. The ulp errors increase significantly when the argument approaches the value of π (or Pi) for FSIN, and when it approaches π/2(or Pi/2) for FCOS, FSINCOS, and FPTAN.

For all three IEEE precisions supported (32-bit single precision, 64-bit double precision, and 80-bit double-extended precision), applying FSIN, FCOS, FSINCOS, or FPTAN to arguments larger than a certain value can lead to reduced arguments (calculated internally) that are inaccurate or even very inaccurate in some cases. This leads to equally inaccurate approximations of the corresponding mathematical functions. In particular, arguments that are close to certain values will lose significance when reduced, leading to increased relative (and ulp) errors in the results of FSIN, FCOS, FSINCOS, and FPTAN. These values are:

- any non-zero multiple of π for FSIN,
- any multiple of π, plus π/2 for FCOS, and
- any non-zero multiple of π/2 for FSINCOS and FPTAN.

If the arguments passed to FSIN, FCOS, FSINCOS, and FPTAN are not close to these values then even the finite approximation Pi of π used internally for argument reduction will allow for results that have good accuracy.

Therefore, in order to avoid such errors it is recommended to perform accurate argument reduction in software, and to apply FSIN, FCOS, FSINCOS, and FPTAN to reduced arguments only. Regardless of the target precision (single, double, or double-extended), it is safe to reduce the argument to a value smaller in absolute value than about 3π/4 for FSIN, and smaller than about 3π/8 for FCOS, FSINCOS, and FPTAN.

The thresholds shown above are not exact. For example, accuracy measurements show that the double-extended precision result of FSIN will not have errors larger than 0.72 ulp for $|x| < 2.82$ (so $|x| < 3\pi/4$ will ensure good accuracy, as $3\pi/4 < 2.82$). On the same interval, double precision results from FSIN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

Likewise, the double-extended precision result of FCOS will not have errors larger than 0.82 ulp for $|x| < 1.31$ (so $|x| < 3\pi/8$ will ensure good accuracy, as $3\pi/8 < 1.31$). On the same interval, double precision results from FCOS will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

FSINCOS behaves similarly to FSIN and FCOS, combined as a pair.

Finally, the double-extended precision result of FPTAN will not have errors larger than 0.78 ulp for $|x| < 1.25$ (so $|x| < 3\pi/8$ will ensure good accuracy, as $3\pi/8 < 1.25$). On the same interval, double precision results from FPTAN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

A recommended alternative in order to avoid the accuracy issues that might be caused by FSIN, FCOS, FSINCOS, and FPTAN, is to use good quality mathematical library implementations of the sin, cos, sincos, and tan functions, for example those from the Intel® Math Library available in the Intel® Compiler.

The instructions FYL2X and FYL2XP1 are two operand instructions and are guaranteed to be within 1 ulp only when y equals 1. When y is not equal to 1, the maximum ulp error is always within 1.35 ulps in round to nearest mode. (For the two operand functions, monotonicity was proved by holding one of the operands constant.)

8.3.11 x87 FPU Control Instructions

The following instructions control the state and modes of operation of the x87 FPU. They also allow the status of the x87 FPU to be examined:

FINIT/FNINIT	Initialize x87 FPU
FLDCW	Load x87 FPU control word
FSTCW/FNSTCW	Store x87 FPU control word
FSTSW/FNSTSW	Store x87 FPU status word
FCLEX/FNCLEX	Clear x87 FPU exception flags
FLDENV	Load x87 FPU environment
FSTENV/FNSTENV	Store x87 FPU environment
FRSTOR	Restore x87 FPU state
FSAVE/FNSAVE	Save x87 FPU state
FINCSTP	Increment x87 FPU register stack pointer
FDECSTP	Decrement x87 FPU register stack pointer
FFREE	Free x87 FPU register
FNOP	No operation
WAIT/FWAIT	Check for and handle pending unmasked x87 FPU exceptions

The FINIT/FNINIT instructions initialize the x87 FPU and its internal registers to default values.

The FLDCW instructions loads the x87 FPU control word register with a value from memory. The FSTCW/FNSTCW and FSTSW/FNSTSW instructions store the x87 FPU control and status words, respectively, in memory (or for an FSTSW/FNSTSW instruction in a general-purpose register).

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions save the x87 FPU environment and state, respectively, in memory. The x87 FPU environment includes all the x87 FPU's control and status registers; the x87 FPU state includes the x87 FPU environment and the data registers in the x87 FPU register stack. (The FSAVE/FNSAVE instruction also initializes the x87 FPU to default values, like the FINIT/FNINIT instruction, after it saves the original state of the x87 FPU.)

The FLDENV and FRSTOR instructions load the x87 FPU environment and state, respectively, from memory into the x87 FPU. These instructions are commonly used when switching tasks or contexts.

The WAIT/FWAIT instructions are synchronization instructions. (They are actually mnemonics for the same opcode.) These instructions check the x87 FPU status word for pending unmasked x87 FPU exceptions. If any pending unmasked x87 FPU exceptions are found, they are handled before the processor resumes execution of the instructions (integer, floating-point, or system instruction) in the instruction stream. The WAIT/FWAIT instructions are provided to allow synchronization of instruction execution between the x87 FPU and the processor's integer unit. See Section 8.6, "x87 FPU Exception Synchronization," for more information on the use of the WAIT/FWAIT instructions.

8.3.12 Waiting vs. Non-waiting Instructions

All of the x87 FPU instructions except a few special control instructions perform a wait operation (similar to the WAIT/FWAIT instructions), to check for and handle pending unmasked x87 FPU floating-point exceptions, before they perform their primary operation (such as adding two floating-point numbers). These instructions are called **waiting** instructions. Some of the x87 FPU control instructions, such as FSTSW/FNSTSW, have both a waiting and a non-waiting version. The waiting version (with the "F" prefix) executes a wait operation before it performs its primary operation; whereas, the non-waiting version (with the "FN" prefix) ignores pending unmasked exceptions.

Non-waiting instructions allow software to save the current x87 FPU state without first handling pending exceptions or to reset or reinitialize the x87 FPU without regard for pending exceptions.

NOTES

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for a non-waiting instruction to be interrupted prior to being executed to handle a pending x87 FPU exception. The circumstances where this can happen and the resulting action of the processor are described in Section D.2.1.3, “No-Wait x87 FPU Instructions Can Get x87 FPU Interrupt in Window.”

When operating a P6 family, Pentium 4, or Intel Xeon processor in MS-DOS compatibility mode, non-waiting instructions can not be interrupted in this way (see Section D.2.2, “MS-DOS* Compatibility Sub-mode in the P6 Family and Pentium® 4 Processors”).

8.3.13 Unsupported x87 FPU Instructions

The Intel 8087 instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM perform no function in the Intel 387 math coprocessor and later IA-32 processors. If these opcodes are detected in the instruction stream, the x87 FPU performs no specific operation and no internal x87 FPU states are affected.

8.4 X87 FPU FLOATING-POINT EXCEPTION HANDLING

The x87 FPU detects the six classes of exception conditions described in Section 4.9, “Overview of Floating-Point Exceptions”:

- Invalid operation (#I), with two subclasses:
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Denormalized operand (#D)
- Divide-by-zero (#Z)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

Each of the six exception classes has a corresponding flag bit in the x87 FPU status word and a mask bit in the x87 FPU control word (see Section 8.1.3, “x87 FPU Status Register,” and Section 8.1.5, “x87 FPU Control Word,” respectively). In addition, the exception summary (ES) flag in the status word indicates when one or more unmasked exceptions has been detected. The stack fault (SF) flag (also in the status word) distinguishes between the two types of invalid-operation exceptions.

The mask bits can be set with FLDCW, FRSTOR, or FXRSTOR; they can be read with either FSTCW/FNSTCW, FSAVE/FNSAVE, or FXSAVE. The flag bits can be read with the FSTSW/FNSTSW, FSAVE/FNSAVE, or FXSAVE instruction.

NOTE

Section 4.9.1, “Floating-Point Exception Conditions,” provides a general overview of how the IA-32 processor detects and handles the various classes of floating-point exceptions. This information pertains to x87 FPU as well as SSE/SSE2/SSE3 extensions.

The following sections give specific information about how the x87 FPU handles floating-point exceptions that are unique to the x87 FPU.

8.4.1 Arithmetic vs. Non-arithmetic Instructions

When dealing with floating-point exceptions, it is useful to distinguish between **arithmetic instructions** and **non-arithmetic instructions**. Non-arithmetic instructions have no operands or do not make substantial changes to their operands. Arithmetic instructions do make significant changes to their operands; in particular, they make changes that could result in floating-point exceptions being signaled. Table 8-9 lists the non-arithmetic and arith-

metic instructions. It should be noted that some non-arithmetic instructions can signal a floating-point stack (fault) exception, but this exception is not the result of an operation on an operand.

Table 8-9. Arithmetic and Non-arithmetic Instructions

Non-arithmetic Instructions	Arithmetic Instructions
FABS	F2XM1
FCHS	FADD/FADDP
FCLEX	FBLD
FDECSTP	FBSTP
FFREE	FCOM/FCOMP/FCOMPP
FINCSTP	FCOS
FINIT/FNINIT	FDIV/FDIVP/FDIVR/FDIVRP
FLD (register-to-register)	FIADD
FLD (extended format from memory)	FICOM/FICOMP
FLD constant	FIDIV/FIDIVR
FLDCW	FILD
FLDENV	FIMUL
FNOP	FIST/FISTP [1]
FRSTOR	FISUB/FISUBR
FSAVE/FNSAVE	FLD (single and double)
FST/FSTP (register-to-register)	FMUL/FMULP
FSTP (extended format to memory)	FPATAN
FSTCW/FNSTCW	FPREM/FPREM1
FSTENV/FNSTENV	FPTAN
FSTSW/FNSTSW	FRNDINT
WAIT/FWAIT	FSCALE
FXAM	FSIN
FXCH	FSINCOS
	FSQRT
	FST/FSTP (single and double)
	FSUB/FSUBP/FSUBR/FSUBRP
	FTST
	FUCOM/FUCOMP/FUCOMPP
	FXTRACT
	FYL2X/FYL2XP1

NOTE:

1. The FISTTP instruction in SSE3 is an arithmetic x87 FPU instruction.

8.5 X87 FPU FLOATING-POINT EXCEPTION CONDITIONS

The following sections describe the various conditions that cause a floating-point exception to be generated by the x87 FPU and the masked response of the x87 FPU when these conditions are detected. *Intel® 64 and IA-32 Archi-*

tectures Software Developer's Manual, Volumes 2A & 2B, list the floating-point exceptions that can be signaled for each floating-point instruction.

See Section 4.9.2, "Floating-Point Exception Priority," for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

8.5.1 Invalid Operation Exception

The floating-point invalid-operation exception occurs in response to two sub-classes of operations:

- Stack overflow or underflow (#IS)
- Invalid arithmetic operand (#IA)

The flag for this exception (IE) is bit 0 of the x87 FPU status word, and the mask bit (IM) is bit 0 of the x87 FPU control word. The stack fault flag (SF) of the x87 FPU status word indicates the type of operation that caused the exception. When the SF flag is set to 1, a stack operation has resulted in stack overflow or underflow; when the flag is cleared to 0, an arithmetic instruction has encountered an invalid operand. Note that the x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. As a result, the state of the SF flag can be 1 following an invalid-arithmetic-operation exception, if it was not cleared from the last time a stack overflow or underflow condition occurred. See Section 8.1.3.4, "Stack Fault Flag," for more information about the SF flag.

8.5.1.1 Stack Overflow or Underflow Exception (#IS)

The x87 FPU tag word keeps track of the contents of the registers in the x87 FPU register stack (see Section 8.1.7, "x87 FPU Tag Word"). It then uses this information to detect two different types of stack faults:

- **Stack overflow** — An instruction attempts to load a non-empty x87 FPU register from memory. A non-empty register is defined as a register containing a zero (tag value of 01), a valid value (tag value of 00), or a special value (tag value of 10).
- **Stack underflow** — An instruction references an empty x87 FPU register as a source operand, including attempting to write the contents of an empty register to memory. An empty register has a tag value of 11.

NOTES

The term stack overflow originates from the situation where the program has loaded (pushed) eight values from memory onto the x87 FPU register stack and the next value pushed on the stack causes a stack wraparound to a register that already contains a value.

The term stack underflow originates from the opposite situation. Here, a program has stored (popped) eight values from the x87 FPU register stack to memory and the next value popped from the stack causes stack wraparound to an empty register.

When the x87 FPU detects stack overflow or underflow, it sets the IE flag (bit 0) and the SF flag (bit 6) in the x87 FPU status word to 1. It then sets condition-code flag C1 (bit 9) in the x87 FPU status word to 1 if stack overflow occurred or to 0 if stack underflow occurred.

If the invalid-operation exception is masked, the x87 FPU returns the floating point, integer, or packed decimal integer indefinite value to the destination operand, depending on the instruction being executed. This value overwrites the destination register or memory location specified by the instruction.

If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software") and the top-of-stack pointer (TOP) and source operands remain unchanged.

8.5.1.2 Invalid Arithmetic Operand Exception (#IA)

The x87 FPU is able to detect a variety of invalid arithmetic operations that can be coded in a program. These operations are listed in Table 8-10. (This list includes the invalid operations defined in IEEE Standard 754.)

When the x87 FPU detects an invalid arithmetic operand, it sets the IE flag (bit 0) in the x87 FPU status word to 1. If the invalid-operation exception is masked, the x87 FPU then returns an indefinite value or QNaN to the destina-

tion operand and/or sets the floating-point condition codes as shown in Table 8-10. If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software") and the top-of-stack pointer (TOP) and source operands remain unchanged.

Table 8-10. Invalid Arithmetic Operations and the Masked Responses to Them

Condition	Masked Response
Any arithmetic operation on an operand that is in an unsupported format.	Return the QNaN floating-point indefinite value to the destination operand.
Any arithmetic operation on a SNaN.	Return a QNaN to the destination operand (see Table 4-7).
Ordered compare and test operations: one or both operands are NaNs.	Set the condition code flags (C0, C2, and C3) in the x87 FPU status word or the CF, PF, and ZF flags in the EFLAGS register to 111B (not comparable).
Addition: operands are opposite-signed infinities. Subtraction: operands are like-signed infinities.	Return the QNaN floating-point indefinite value to the destination operand.
Multiplication: ∞ by 0; 0 by ∞.	Return the QNaN floating-point indefinite value to the destination operand.
Division: ∞ by ∞; 0 by 0.	Return the QNaN floating-point indefinite value to the destination operand.
Remainder instructions FPREM, FPREM1: modulus (divisor) is 0 or dividend is ∞.	Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.
Trigonometric instructions FCOS, FPTAN, FSIN, FSINCOS: source operand is ∞.	Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.
FSQRT: negative operand (except FSQRT (–0) = –0); FYL2X: negative operand (except FYL2X (–0) = –∞); FYL2XP1: operand more negative than –1.	Return the QNaN floating-point indefinite value to the destination operand.
FBSTP: Converted value cannot be represented in 18 decimal digits, or source value is an SNaN, QNaN, $\pm\infty$, or in an unsupported format.	Store packed BCD integer indefinite value in the destination operand.
FIST/FISTP: Converted value exceeds representable integer range of the destination operand, or source value is an SNaN, QNaN, ±∞, or in an unsupported format.	Store integer indefinite value in the destination operand.
FXCH: one or both registers are tagged empty.	Load empty registers with the QNaN floating-point indefinite value, then perform the exchange.

Normally, when one or both of the source operands is a QNaN (and neither is an SNaN or in an unsupported format), an invalid-operand exception is not generated. An exception to this rule is most of the compare instructions (such as the FCOM and FCOMI instructions) and the floating-point to integer conversion instructions (FIST/FISTP and FBSTP). With these instructions, a QNaN source operand will generate an invalid-operand exception.

8.5.2 Denormal Operand Exception (#D)

The x87 FPU signals the denormal-operand exception under the following conditions:

- If an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers").
- If an attempt is made to load a denormal single-precision or double-precision floating-point value into an x87 FPU register. (If the denormal value being loaded is a double extended-precision floating-point value, the denormal-operand exception is not reported.)

The flag (DE) for this exception is bit 1 of the x87 FPU status word, and the mask bit (DM) is bit 1 of the x87 FPU control word.

When a denormal-operand exception occurs and the exception is masked, the x87 FPU sets the DE flag, then proceeds with the instruction. The denormal operand in single- or double-precision floating-point format is automatically normalized when converted to the double extended-precision floating-point format. Subsequent operations will benefit from the additional precision of the internal double extended-precision floating-point format.

When a denormal-operand exception occurs and the exception is not masked, the DE flag is set and a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”). The top-of-stack pointer (TOP) and source operands remain unchanged.

For additional information about the denormal-operation exception, see Section 4.9.1.2, “Denormal Operand Exception (#D).”

8.5.3 Divide-By-Zero Exception (#Z)

The x87 FPU reports a floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) for this exception is bit 2 of the x87 FPU status word, and the mask bit (ZM) is bit 2 of the x87 FPU control word. The FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, and FIDIVR instructions and the other instructions that perform division internally (FYL2X and FXTRACT) can report the divide-by-zero exception.

When a divide-by-zero exception occurs and the exception is masked, the x87 FPU sets the ZE flag and returns the values shown in Table 8-10. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”), and the top-of-stack pointer (TOP) and source operands remain unchanged.

Table 8-11. Divide-By-Zero Conditions and the Masked Responses to Them

Condition	Masked Response
Divide or reverse divide operation with a 0 divisor.	Returns an ∞ signed with the exclusive OR of the sign of the two operands to the destination operand.
FYL2X instruction.	Returns an ∞ signed with the opposite sign of the non-zero operand to the destination operand.
FXTRACT instruction.	ST(1) is set to –∞; ST(0) is set to 0 with the same sign as the source operand.

8.5.4 Numeric Overflow Exception (#O)

The x87 FPU reports a floating-point numeric overflow exception (#O) whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the floating-point format of the destination operand. (See Section 4.9.1.4, “Numeric Overflow Exception (#O),” for additional information about the numeric overflow exception.)

When using the x87 FPU, numeric overflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (using the FST and FSTP instructions), where a within-range value in a data register is stored in memory in a single-precision or double-precision floating-point format. The numeric overflow exception cannot occur when storing values in an integer or BCD integer format. Instead, the invalid-arithmetic-operand exception is signaled.

The flag (OE) for the numeric-overflow exception is bit 3 of the x87 FPU status word, and the mask bit (OM) is bit 3 of the x87 FPU control word.

When a numeric-overflow exception occurs and the exception is masked, the x87 FPU sets the OE flag and returns one of the values shown in Table 4-10. The value returned depends on the current rounding mode of the x87 FPU (see Section 8.1.5.3, “Rounding Control Field”).

The action that the x87 FPU takes when numeric overflow occurs and the numeric-overflow exception is not masked, depends on whether the instruction is supposed to store the result in memory or on the register stack.

- **Destination is a memory location** — The OE flag is set and a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”). The top-of-stack pointer (TOP) and source and destination operands remain unchanged. Because the data in the stack is in double extended-precision format,

the exception handler has the option either of re-executing the store instruction after proper adjustment of the operand or of rounding the significand on the stack to the destination's precision as the standard requires. The exception handler should ultimately store a value into the destination location in memory if the program is to continue.

- **Destination is the register stack** — The significand of the result is rounded according to current settings of the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted by dividing it by 2^{24576}. (For instructions not affected by the precision field, the significand is rounded to double-extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in the x87 FPU status word (called in this situation the "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the OE flag is set and a software exception handler is invoked. The scaling bias value 24,576 is equal to $3 * 2^{13}$. Biasing the exponent by 24,576 normally translates the number as nearly as possible to the middle of the double extended-precision floating-point exponent range so that, if desired, it can be used in subsequent scaled operations with less risk of causing further exceptions.

  When using the FSCALE instruction, massive overflow can occur, where the result is too large to be represented, even with a bias-adjusted exponent. Here, if overflow occurs again, after the result has been biased, a properly signed ∞ is stored in the destination operand.

8.5.5 Numeric Underflow Exception (#U)

The x87 FPU detects a potential floating-point numeric underflow condition whenever the result of an arithmetic instruction is non-zero and tiny; that is, the magnitude of the rounded result with unbounded exponent is non-zero and less than the smallest possible normalized, finite value that will fit into the floating-point format of the destination operand. (See Section 4.9.1.5, "Numeric Underflow Exception (#U)," for additional information about the numeric underflow exception.)

Like numeric overflow, numeric underflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (with the FST and FSTP instructions), where a within-range value in a data register is stored in memory in the smaller single-precision or double-precision floating-point formats. A numeric underflow exception cannot occur when storing values in an integer or BCD integer format, because a value with magnitude less than 1 is always rounded to an integral value of 0 or 1, depending on the rounding mode in effect.

The flag (UE) for the numeric-underflow exception is bit 4 of the x87 FPU status word, and the mask bit (UM) is bit 4 of the x87 FPU control word.

When a numeric-underflow condition occurs and the exception is masked, the x87 FPU performs the operation described in Section 4.9.1.5, "Numeric Underflow Exception (#U)."

When the exception is not masked, the action of the x87 FPU depends on whether the instruction is supposed to store the result in a memory location or on the x87 FPU resister stack.

- **Destination is a memory location** — (Can occur only with a store instruction.) The UE flag is set and a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software"). The top-of-stack pointer (TOP) and source and destination operands remain unchanged, and no result is stored in memory.
  Because the data in the stack is in double extended-precision format, the exception handler has the option either of re-exchanges the store instruction after proper adjustment of the operand or of rounding the significand on the stack to the destination's precision as the standard requires. The exception handler should ultimately store a value into the destination location in memory if the program is to continue.
- **Destination is the register stack** — The significand of the result is rounded according to current settings of the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted by multiplying it by 2^{24576}. (For instructions not affected by the precision field, the significand is rounded to double extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in the x87 FPU status register (acting here as a "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the UE flag is set and a software exception handler is invoked. The scaling bias value 24,576 is the same as is used for the overflow exception and has the same effect, which is to translate the result as nearly as possible to the middle of the double extended-precision floating-point exponent range.

When using the FSCALE instruction, massive underflow can occur, where the magnitude of the result is too small to be represented, even with a bias-adjusted exponent. Here, if underflow occurs again after the result has been biased, a properly signed 0 is stored in the destination operand.

8.5.6 Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. (See Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)," for additional information about the numeric overflow exception.) Note that the transcendental instructions (FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) by nature produce inexact results.

The inexact-result exception flag (PE) is bit 5 of the x87 FPU status word, and the mask bit (PM) is bit 5 of the x87 FPU control word.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the x87 FPU handles the exception as describe in Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)," with one additional action. The C1 (round-up) bit in the x87 FPU status word is set to indicate whether the inexact result was rounded up (C1 is set) or "not rounded up" (C1 is cleared). In the "not rounded up" case, the least-significant bits of the inexact result are truncated so that the result fits in the destination format.

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the x87 FPU handles the exception as described in the previous paragraph and, in addition, invokes a software exception handler.

If an inexact result occurs in conjunction with numeric overflow or underflow, the x87 FPU carries out one of the following operations:

- If an inexact result occurs in conjunction with masked overflow or underflow, the OE or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions (see Section 8.5.4, "Numeric Overflow Exception (#O)," or Section 8.5.5, "Numeric Underflow Exception (#U)"). If the inexact result exception is unmasked, the x87 FPU also invokes a software exception handler.
- If an inexact result occurs in conjunction with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions (see Section 8.5.4, "Numeric Overflow Exception (#O)," or Section 8.5.5, "Numeric Underflow Exception (#U)") and a software exception handler is invoked.

If an unmasked numeric overflow or underflow exception occurs and the destination operand is a memory location (which can happen only for a floating-point store), the inexact-result condition is not reported and the C1 flag is cleared.

8.6 X87 FPU EXCEPTION SYNCHRONIZATION

Because the integer unit and x87 FPU are separate execution units, it is possible for the processor to execute floating-point, integer, and system instructions concurrently. No special programming techniques are required to gain the advantages of concurrent execution. (Floating-point instructions are placed in the instruction stream along with the integer and system instructions.) However, concurrent execution can cause problems for floating-point exception handlers.

This problem is related to the way the x87 FPU signals the existence of unmasked floating-point exceptions. (Special exception synchronization is not required for masked floating-point exceptions, because the x87 FPU always returns a masked result to the destination operand.)

When a floating-point exception is unmasked and the exception condition occurs, the x87 FPU stops further execution of the floating-point instruction and signals the exception event. On the next occurrence of a floating-point instruction or a WAIT/FWAIT instruction in the instruction stream, the processor checks the ES flag in the x87 FPU status word for pending floating-point exceptions. If floating-point exceptions are pending, the x87 FPU makes an implicit call (traps) to the floating-point software exception handler. The exception handler can then execute recovery procedures for selected or all floating-point exceptions.

Synchronization problems occur in the time between the moment when the exception is signaled and when it is actually handled. Because of concurrent execution, integer or system instructions can be executed during this time. It is thus possible for the source or destination operands for a floating-point instruction that faulted to be overwritten in memory, making it impossible for the exception handler to analyze or recover from the exception.

To solve this problem, an exception synchronizing instruction (either a floating-point instruction or a WAIT/FWAIT instruction) can be placed immediately after any floating-point instruction that might present a situation where state information pertaining to a floating-point exception might be lost or corrupted. Floating-point instructions that store data in memory are prime candidates for synchronization. For example, the following three lines of code have the potential for exception synchronization problems:

```
FILD COUNT          ;Floating-point instruction
INC COUNT           ;Integer instruction
FSQRT               ;Subsequent floating-point instruction
```

In this example, the INC instruction modifies the source operand of the floating-point instruction, FILD. If an exception is signaled during the execution of the FILD instruction, the INC instruction would be allowed to overwrite the value stored in the COUNT memory location before the floating-point exception handler is called. With the COUNT variable modified, the floating-point exception handler would not be able to recover from the error.

Rearranging the instructions, as follows, so that the FSQRT instruction follows the FILD instruction, synchronizes floating-point exception handling and eliminates the possibility of the COUNT variable being overwritten before the floating-point exception handler is invoked.

```
FILD COUNT      ;Floating-point instruction
FSQRT           ;Subsequent floating-point instruction synchronizes
                ;any exceptions generated by the FILD instruction.
INC COUNT       ;Integer instruction
```

The FSQRT instruction does not require any synchronization, because the results of this instruction are stored in the x87 FPU data registers and will remain there, undisturbed, until the next floating-point or WAIT/FWAIT instruction is executed. To absolutely insure that any exceptions emanating from the FSQRT instruction are handled (for example, prior to a procedure call), a WAIT instruction can be placed directly after the FSQRT instruction.

Note that some floating-point instructions (non-waiting instructions) do not check for pending unmasked exceptions (see Section 8.3.11, "x87 FPU Control Instructions"). They include the FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX instructions. When an FNINIT, FNSTENV, FNSAVE, or FNCLEX instruction is executed, all pending exceptions are essentially lost (either the x87 FPU status register is cleared or all exceptions are masked). The FNSTSW and FNSTCW instructions do not check for pending interrupts, but they do not modify the x87 FPU status and control registers. A subsequent "waiting" floating-point instruction can then handle any pending exceptions.

8.7 HANDLING X87 FPU EXCEPTIONS IN SOFTWARE

The x87 FPU in Pentium and later IA-32 processors provides two different modes of operation for invoking a software exception handler for floating-point exceptions: native mode and MS-DOS compatibility mode. The mode of operation is selected by CR0.NE[bit 5]. (See Chapter 2, "System Architecture Overview," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* for more information about the NE flag.)

8.7.1 Native Mode

The native mode for handling floating-point exceptions is selected by setting CR0.NE[bit 5] to 1. In this mode, if the x87 FPU detects an exception condition while executing a floating-point instruction and the exception is unmasked (the mask bit for the exception is cleared), the x87 FPU sets the flag for the exception and the ES flag in the x87 FPU status word. It then invokes the software exception handler through the floating-point-error exception (#MF, exception vector 16), immediately before execution of any of the following instructions in the processor's instruction stream:

- The next floating-point instruction, unless it is one of the non-waiting instructions (FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, and FNSAVE).

- The next WAIT/FWAIT instruction.
- The next MMX instruction.

If the next floating-point instruction in the instruction stream is a non-waiting instruction, the x87 FPU executes the instruction without invoking the software exception handler.

8.7.2 MS-DOS* Compatibility Sub-mode

If CR0.NE[bit 5] is 0, the MS-DOS compatibility mode for handling floating-point exceptions is selected. In this mode, the software exception handler for floating-point exceptions is invoked externally using the processor's FERR#, INTR, and IGNNE# pins. This method of reporting floating-point errors and invoking an exception handler is provided to support the floating-point exception handling mechanism used in PC systems that are running the MS-DOS or Windows* 95 operating system.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems, this approach also limits newer processors to operate with one logical processor active.

The MS-DOS compatibility mode is typically used as follows to invoke the floating-point exception handler:

1. If the x87 FPU detects an unmasked floating-point exception, it sets the flag for the exception and the ES flag in the x87 FPU status word.
2. If the IGNNE# pin is deasserted, the x87 FPU then asserts the FERR# pin either immediately, or else delayed (deferred) until just before the execution of the next waiting floating-point instruction or MMX instruction. Whether the FERR# pin is asserted immediately or delayed depends on the type of processor, the instruction, and the type of exception.
3. If a preceding floating-point instruction has set the exception flag for an unmasked x87 FPU exception, the processor freezes just before executing the next WAIT instruction, waiting floating-point instruction, or MMX instruction. Whether the FERR# pin was asserted at the preceding floating-point instruction or is just now being asserted, the freezing of the processor assures that the x87 FPU exception handler will be invoked before the new floating-point (or MMX) instruction gets executed.
4. The FERR# pin is connected through external hardware to IRQ13 of a cascaded, programmable interrupt controller (PIC). When the FERR# pin is asserted, the PIC is programmed to generate an interrupt 75H.
5. The PIC asserts the INTR pin on the processor to signal the interrupt 75H.
6. The BIOS for the PC system handles the interrupt 75H by branching to the interrupt 02H (NMI) interrupt handler.
7. The interrupt 02H handler determines if the interrupt is the result of an NMI interrupt or a floating-point exception.
8. If a floating-point exception is detected, the interrupt 02H handler branches to the floating-point exception handler.

If the IGNNE# pin is asserted, the processor ignores floating-point error conditions. This pin is provided to inhibit floating-point exceptions from being generated while the floating-point exception handler is servicing a previously signaled floating-point exception.

Appendix D, "Guidelines for Writing x87 FPU Exception Handlers," describes the MS-DOS compatibility mode in much greater detail. This mode is somewhat more complicated in the Intel486 and Pentium processor implementations, as described in Appendix D.

8.7.3 Handling x87 FPU Exceptions in Software

Section 4.9.3, "Typical Actions of a Floating-Point Exception Handler," shows actions that may be carried out by a floating-point exception handler. The state of the x87 FPU can be saved with the FSTENV/FNSTENV or FSAVE/FNSAVE instructions (see Section 8.1.10, "Saving the x87 FPU's State with FSTENV/FNSTENV and FSAVE/FNSAVE").

If the faulting floating-point instruction is followed by one or more non-floating-point instructions, it may not be useful to re-execute the faulting instruction. See Section 8.6, “x87 FPU Exception Synchronization,” for more information on synchronizing floating-point exceptions.

In cases where the handler needs to restart program execution with the faulting instruction, the IRET instruction cannot be used directly. The reason for this is that because the exception is not generated until the next floating-point or WAIT/FWAIT instruction following the faulting floating-point instruction, the return instruction pointer on the stack may not point to the faulting instruction. To restart program execution at the faulting instruction, the exception handler must obtain a pointer to the instruction from the saved x87 FPU state information, load it into the return instruction pointer location on the stack, and then execute the IRET instruction.

See Section D.3.4, “x87 FPU Exception Handling Examples,” for general examples of floating-point exception handlers and for specific examples of how to write a floating-point exception handler when using the MS-DOS compatibility mode.

3. Updates to Chapter 1, Volume 2

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

Change to this chapter: Added reference to new volume 4: Model Specific Registers.

CHAPTER 1
ABOUT THIS MANUAL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 NOTATIONAL CONVENTIONS

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

1.3.1 Bit and Byte Order

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

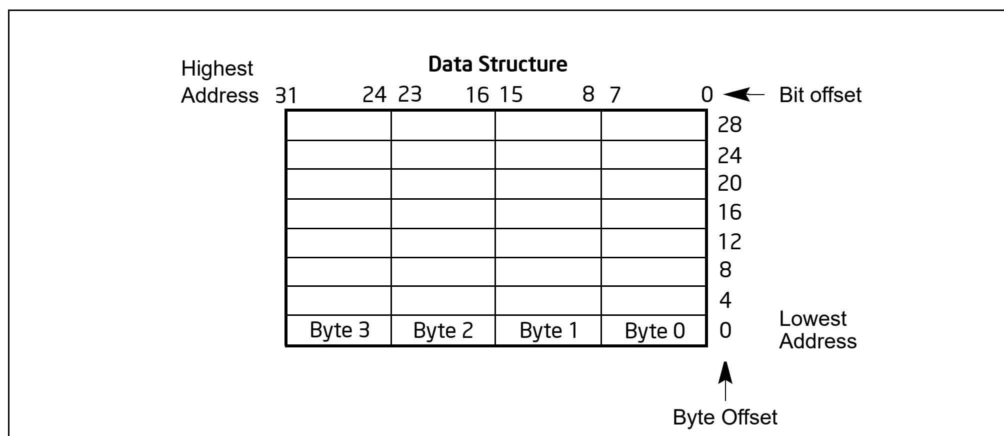


Figure 1-1. Bit and Byte Order

1.3.2 Reserved Bits and Software Compatibility

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

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

NOTE

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

1.3.3 Instruction Operands

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

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

where:

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

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

For example:

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

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

1.3.4 Hexadecimal and Binary Numbers

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

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

1.3.5 Segmented Addressing

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

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

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

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

```
DS:FF79H
```

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

```
CS:EIP
```

1.3.6 Exceptions

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

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

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

```
#GP(0)
```

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

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

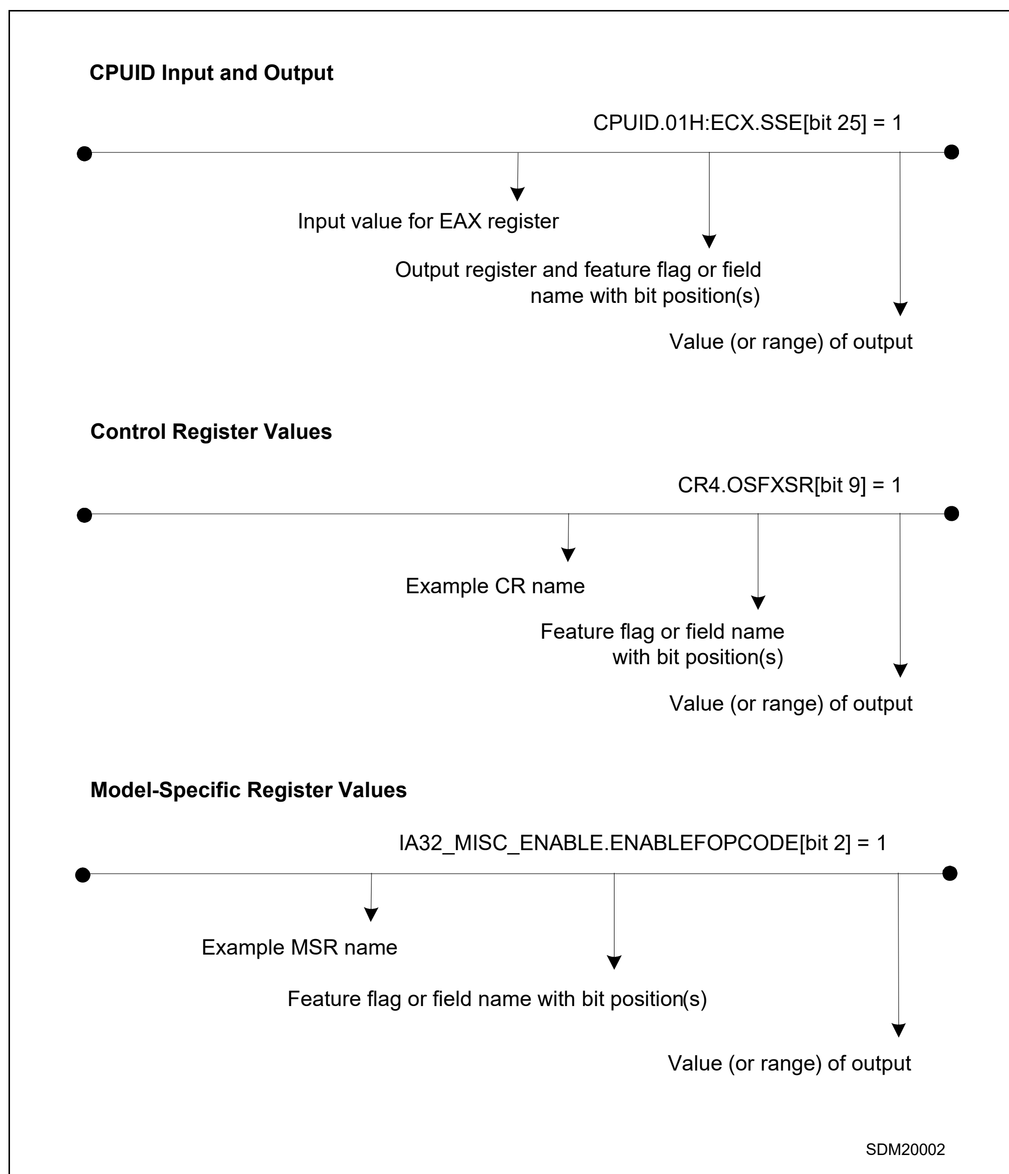


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.4 RELATED LITERATURE

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

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

See also:

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

- Intel® Software Development Tools:
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- Intel 64 Architecture x2APIC Specification:

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

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

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

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

More relevant links are:

- Intel® Developer Zone:

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

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

  http://www.intel.com/support/processors/
- Software products and packages:

  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® Hyper-Threading Technology (Intel® HT Technology):

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

4. Updates to Chapter 3, Volume 2A

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

Change to this chapter: Updates to the following instructions: ADD, ADDPS, ADDSD, ADDSS, ANDNPS, ANDPS, BNDLDX, BNDMK, BNDSTX, BOUND, CALL, CLAC, CLFLUSH, CMPPS, CMPSD, CMPSS, COMISD, COMISS, CPUID, CVTDQ2PD, CVTDQ2PS, CVTPI2PS, CVTPS2PD, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS,CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI, DIVPS, DIVSD, DIVSS, EMMS, FXRSTOR/FXRSTOR64, FXSAVE/FXSAVE64, INT n/INTO/INT 3, IRET/IRETD, LAR, LDMXCSR, LFENCE, LGDT/LIDT, LMSW.

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-L) in IA-32e, protected, virtual-8086, and real-address modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX and system instructions. See also Chapter 4, “Instruction Set Reference, M-U,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, and Chapter 5, “Instruction Set Reference, V-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

3.1.1 Instruction Format

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

CMC—Complement Carry Flag [this is an example]

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F5	CMC	ZO	V/V	NA	Complement carry flag.

Instruction Operand Encoding

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

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

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

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r —** Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct —** A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io —** A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro —** Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- +**i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5
SIL	Yes	6	SI	None	6	ESI	None	6	RSI	None	6
DIL	Yes	7	DI	None	7	EDI	None	7	RDI	None	7
Registers R8 - R15 (see below): Available in 64-Bit Mode Only											

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
R8L	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9L	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10L	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11L	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12L	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13L	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14L	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15L	Yes	7	R15W	Yes	7	R15D	Yes	7	R15	Yes	7

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

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

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

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

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

  — **NDS, NDD, DDS:** Specifies that VEX.vvvv field is valid for the encoding of a register operand:

  - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
  - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
  - VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
  - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.

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

  - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS).
  - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distin-

guished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either “SS“, “SD“, or “SI“ for SIMD floating-point conversion instructions.

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

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

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

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

— **W0:** VEX.W=0.

— **W1:** VEX.W=1.

— The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.

— **WIG:** can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.

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

- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].
- In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

EVEX.[NDS/NDD/DDS].[128,256,512,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [ib]

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.6.1 for more detail on the EVEX prefix.

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

— **NDS, NDD, DDS:** implies that EVEX.vvvv (and EVEX.v′) field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v′ must be 1b).

— **128, 256, 512, LIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.

— **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.

— **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.

— **W0:** EVEX.W=0.

— **W1:** EVEX.W=1.

— **WIG:** EVEX.W bit ignored

- **opcode** — Instruction opcode.
- In general, the encoding of EVEX.R and R′, EVEX.X and X′, and EVEX.B and B′ fields are not shown explicitly in the opcode column.

3.1.1.3 Instruction Column in the Opcode Summary Table

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

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The ith element from the top of the FPU register stack ($i \leftarrow 0$ through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32**— An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **<XMM0>**— Indicates implied use of the XMM0 register.

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

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.
- **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.
- **ymm/m256** — A YMM register or 256-bit memory operand.
- **<YMM0>**— Indicates use of the YMM0 register as an implicit argument.
- **bnd** — A 128-bit bounds register. BND0 through BND3.
- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — A ZMM register or 512-bit memory operand.
- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — A vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.

- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.
- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — the destination in an instruction. This field is encoded by reg_field.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

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

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tupletypes. The Op/En column of an EVEX encoded instruction uses an abbreviation that corresponds to the tupletype abbreviation (and may include an additional abbreviation related to ModR/M and vvvv encoding). Most EVEX encoded instructions with VEX encoded equivalent have the ModR/M and vvvv encoding order. In such cases, the Tuple abbreviation is shown and the ModR/M, vvvv encoding abbreviation may be omitted.

NOTES

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

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

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

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

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

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

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

3.1.1.6 CPUID Support Column in the Instruction Summary Table

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

3.1.1.7 Description Column in the Instruction Summary Table

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

3.1.1.8 Description Section

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

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

- **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm_field** — shorthand for the ModR/M *r/m* field and any REX.B
- **reg_field** — shorthand for the ModR/M *reg* field and any REX.R

3.1.1.9 Operation Section

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

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A ← B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

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

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

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **VLMAX** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's prefix but is instead determined by the current value of XCR0. For existing processors, VLMAX is 256 whenever XCR0.YMM[bit 2] is 1. Future processors may defined new bits in XCR0 whose setting may imply other values for VLMAX.

VLMAX Definition

XCR0 Component	VLMAX
XCR0.YMM	256

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword or Quadword Onto the Stack" section in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **Pop()** — removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value from the Stack" section of Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

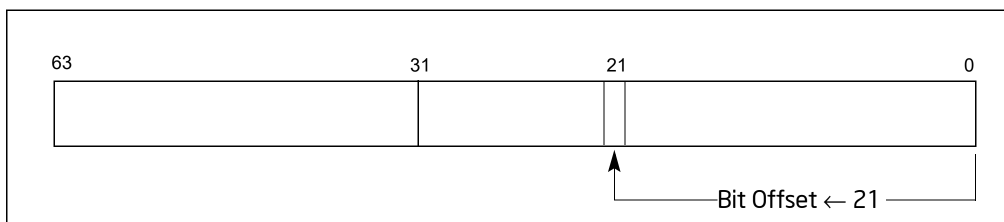


Figure 3-1. Bit Offset for BIT[RAX, 21]

If BitBase is a memory address, the BitOffset can range has different ranges depending on the operand size (see Table 3-2).

Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	-2^{15} to $2^{15}-1$
32	0 to 31	-2^{31} to $2^{31}-1$
64	0 to 63	-2^{63} to $2^{63}-1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

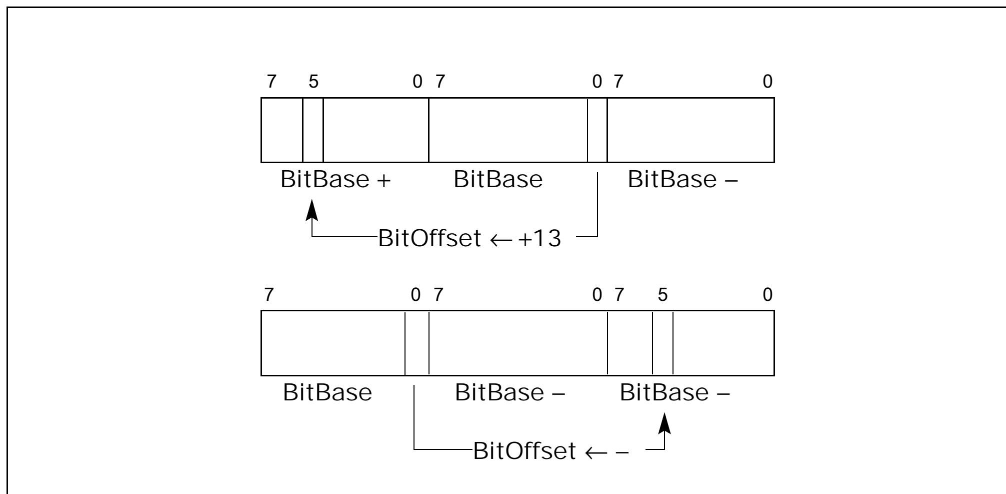


Figure 3-2. Memory Bit Indexing

3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- __m128, __m256 and __m512 can represent 4, 8 or 16 packed single-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single-precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128d, __m256d and __m512d can represent 2, 4 or 8 packed double-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double-precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128i, __m256i and __m512i can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.

Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single-precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- _mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- _mm256_ indicates the function operates on 256-bit vectors.
- _mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single-precision floating-point data. Packed single-precision floating-point data corresponds to arrays of the C/C++ type *float* with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the *_mm_loadu_ps, _mm256_loadu_ps,* or *_mm512_loadu_ps* functions, or created from individual values using *_mm_set_ps, _mm256_set_ps,* or *_mm512_set_ps* functions, and they can be stored in an array using *_mm_storeu_ps, _mm256_storeu_ps,* or *_mm512_storeu_ps.*
- _ss, which indicates a function that operates on scalar single-precision floating-point data. Single-precision floating-point data corresponds to the C/C++ type *float*, and values of type float can be converted to type __m128 for use with these functions using the *_mm_set_ss* function, and converted back using the *_mm_cvtss_f32* function. When used with functions that operate on packed single-precision floating-point data the scalar element corresponds with the first packed value.
- _pd, which indicates a function that operates on packed double-precision floating-point data. Packed double-precision floating-point data corresponds to arrays of the C/C++ type *double* with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the *_mm_loadu_pd, _mm256_loadu_pd,* or *_mm512_loadu_pd* functions, or created from individual values using *_mm_set_pd, _mm2566_set_pd,* or *_mm512_set_pd* functions, and they can be stored in an array using *_mm_storeu_pd, _mm256_storeu_pd,* or *_mm512_storeu_pd.*
- _sd, which indicates a function that operates on scalar double-precision floating-point data. Double-precision floating-point data corresponds to the C/C++ type *double*, and values of type double can be converted to type __m128d for use with these functions using the *_mm_set_sd* function, and converted back using the *_mm_cvtsd_f64* function. When used with functions that operate on packed double-precision floating-point data the scalar element corresponds with the first packed value.
- _epi8, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of *signed char* with 16, 32 or 64 elements. Values of this type can be created from individual elements using *_mm_set_epi8, _mm256_set_epi8,* or *_mm512_set_epi8* functions.
- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of *short* with 8, 16 or 32 elements. Values of this type can be created from individual elements using *_mm_set_epi16, _mm256_set_epi16,* or *_mm512_set_epi16* functions.
- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of *int* with 4, 8 or 16 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of *long long* (or *long* if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of *unsigned char* with 16, 32 or 64 elements.

- _epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of *unsigned short* with 8, 16 or 32 elements.
- _epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of *unsigned* with 4, 8 or 16 elements.
- _epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of *unsigned long long* (or *unsigned long* if it is a 64-bit data type) with 2, 4 or 8 elements.
- _si128, which indicates a function that operates on a single 128-bit value of type __m128i.
- _si256, which indicates a function that operates on a single a 256-bit value of type __m256i.
- _si512, which indicates a function that operates on a single a 512-bit value of type __m512i.

Values of any packed integer type can be loaded from an array using the *_mm_loadu_si128, _mm256_loadu_si256,* or *_mm512_loadu_si512* functions, and they can be stored in an array using *_mm_storeu_si128, _mm256_storeu_si256,* or *_mm512_storeu_si512*.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as __int64. The C Standard header *stdint.h* defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

Table 3-3. Standard and Non-standard Data Types

Non-standard Type	Standard Type (from stdint.h)
__int64	int64_t
unsigned __int64	uint64_t
__int32	int32_t
unsigned __int32	uint32_t
__int16	int16_t
unsigned __int16	uint16_t

For a more detailed description of each intrinsic function and additional information related to its usage, refer to the online Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide.

3.1.1.11 Flags Affected Section

The "Flags Affected" section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, "EFLAGS Cross-Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Non-conventional assignments are described in the "Operation" section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an "FPU Flags Affected" section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The "Protected Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound

sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, "Procedure Calls, Interrupts, and Exceptions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

Table 3-4. Intel 64 and IA-32 General Exceptions

Vector	Name	Source	Protected Mode [1]	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT 3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	Reserved	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	Reserved	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection [2]	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	Reserved	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	Reserved	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes

NOTES:

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

3.1.1.14 Real-Address Mode Exceptions Section

The "Real-Address Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

3.1.1.15 Virtual-8086 Mode Exceptions Section

The "Virtual-8086 Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).

3.1.1.16 Floating-Point Exceptions Section

The "Floating-Point Exceptions" section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See "Floating-Point Exception Conditions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a detailed description of these exceptions.

Table 3-5. x87 FPU Floating-Point Exceptions

Mnemonic	Name	Source
#IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

3.1.1.17 SIMD Floating-Point Exceptions Section

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to "SSE and SSE2 Exceptions", in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Table 3-6. SIMD Floating-Point Exceptions

Mnemonic	Name	Source
#I	Floating-point invalid operation	Invalid arithmetic operation or source operand
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result	Inexact result (precision)

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m8.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /*r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /*r*	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /*r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /*r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /*r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /*r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /*r*	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /*r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /*r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /*r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the CF and OF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

DEST ← DEST + SRC;

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	RM	V/V	SSE	Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Add four, eight or sixteen packed single-precision floating-point values from the first source operand with the second source operand, and stores the packed single-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VADDPS (EVEX encoded versions) when src2 operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
```

```
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VADDPS (EVEX encoded versions) when src2 operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VADDPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[159:128] ← SRC1[159:128] + SRC2[159:128]
DEST[191:160]← SRC1[191:160] + SRC2[191:160]
DEST[223:192] ← SRC1[223:192] + SRC2[223:192]
DEST[255:224] ← SRC1[255:224] + SRC2[255:224].
DEST[MAX_VL-1:256] ← 0

VADDPS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

ADDPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VADDPS __m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);
VADDPS __m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VADDPS __m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);
VADDPS __m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);
VADDPS __m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);
VADDPS __m512 _mm512_add_round_ps (__m512 a, __m512 b, int);
VADDPS __m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VADDPS __m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);
ADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	RM	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

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

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

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
```

DEST[MAX_VL-1:128] ← 0

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

ADDSD (128-bit Legacy SSE version)
DEST[63:0] ←DEST[63:0] + SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	RM	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

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

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

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSS (EVEX encoded versions)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
         SET_RM(EVEX.RC);
    ELSE
         SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN     DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
    ELSE
         IF *merging-masking*                    ; merging-masking
              THEN *DEST[31:0] remains unchanged*
              ELSE                               ; zeroing-masking
                   THEN DEST[31:0] ← 0
         FI;
FI;
DEST[127:32] ← SRC1[127:32]
```

DEST[MAX_VL-1:128] ← 0

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDNPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VANDNPS (VEX.256 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160] ← (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] ← (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] ← (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].
DEST[MAX_VL-1:256] ← 0
```

VANDNPS (VEX.128 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAX_VL-1:128] ← 0
```

ANDNPS (128-bit Legacy SSE version)

```
DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 54 /r ANDPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
                IF (EVEX.b == 1) AND (SRC2 *is memory*)
                    THEN
                        DEST[i+63:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]
                    ELSE
                        DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
                FI;
        ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;
```

VANDPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE AND SRC2[255:224].
DEST[MAX_VL-1:256] ← 0;
```

VANDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[MAX_VL-1:128] ← 0;
```

ANDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m512 _mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 _mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 _mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 _mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 _mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 _mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 _mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	NA

Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;
```

Outside 64-bit mode

```
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 );
Temp_lb[31:0] ← LoadFrom(A_BTE);
Temp_ub[31:0] ← LoadFrom(A_BTE + 4);
Temp_ptr[31:0] ← LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
```

```
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

In 64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:20] «12 );1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] ← (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 );
Temp_lb[63:0] ← LoadFrom(A_BTE);
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.

BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1B /r BNDMK bnd, m32	RM	NE/V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 0F 1B /r BNDMK bnd, m64	RM	V/NE	MPX	Make lower and upper bounds from m64 and store them in bnd.

Instruction Operand Encoding

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

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

Operation

```
BND.LB ← SRCMEM.base;
IF 64-bit mode Then
    BND.UB ← NOT(LEA.64_bits(SRCMEM));
ELSE
    BND.UB ← Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
	If RIP-relative addressing is used.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	NA

Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;
```

Outside 64-bit mode

```
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 ); // address of Bound table entry
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
```

In 64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:20] «12 );1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] ← (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 ); // address of Bound table entry
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
62 /*r*	BOUND *r16, m16&16*	RM	Invalid	Valid	Check if *r16* (array index) is within bounds specified by *m16&16.*
62 /*r*	BOUND *r32, m32&32*	RM	Invalid	Valid	Check if *r32* (array index) is within bounds specified by *m32&32.*

Instruction Operand Encoding

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

Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64bit Mode
    THEN
        #UD;
    ELSE
        IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN
        (* Below lower bound or above upper bound *)
            IF <equation for PL enabled> THEN BNDSTATUS ← 0
            #BR;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 *cw*	CALL *rel16*	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 *cd*	CALL *rel32*	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL *r/m16*	M	N.E.	Valid	Call near, absolute indirect, address given in *r/m16.*
FF /2	CALL *r/m32*	M	N.E.	Valid	Call near, absolute indirect, address given in *r/m32.*
FF /2	CALL *r/m64*	M	Valid	N.E.	Call near, absolute indirect, address given in *r/m*64.
9A *cd*	CALL *ptr16:16*	D	Invalid	Valid	Call far, absolute, address given in operand.
9A *cp*	CALL *ptr16:32*	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL *m16:16*	M	Valid	Valid	Call far, absolute indirect address given in *m16:16.* In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL *m16:32*	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL *m16:64*	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32, or r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand- size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Operation

```
IF near call
   THEN IF near relative call
       THEN
           IF OperandSize = 64
               THEN
                   tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
                   tempRIP ← RIP + tempDEST;
                   IF stack not large enough for a 8-byte return address
                       THEN #SS(0); FI;
                   Push(RIP);
                   RIP ← tempRIP;
           FI;
           IF OperandSize = 32
               THEN
                   tempEIP ← EIP + DEST; (* DEST is rel32 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
                   IF stack not large enough for a 4-byte return address
                       THEN #SS(0); FI;
                   Push(EIP);
                   EIP ← tempEIP;
           FI;
           IF OperandSize = 16
               THEN
                   tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
                   IF stack not large enough for a 2-byte return address
                       THEN #SS(0); FI;
                   Push(IP);
                   EIP ← tempEIP;
           FI;
       ELSE (* Near absolute call *)
           IF OperandSize = 64
               THEN
                   tempRIP ← DEST; (* DEST is r/m64 *)
                   IF stack not large enough for a 8-byte return address
                       THEN #SS(0); FI;
                   Push(RIP);
                   RIP ← tempRIP;
           FI;
           IF OperandSize = 32
               THEN
                   tempEIP ← DEST; (* DEST is r/m32 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
                   IF stack not large enough for a 4-byte return address
                       THEN #SS(0); FI;
                   Push(EIP);
                   EIP ← tempEIP;
           FI;
           IF OperandSize = 16
               THEN
                   tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
```

```
                        IF stack not large enough for a 2-byte return address
                             THEN #SS(0); FI;
                        Push(IP);
                        EIP ← tempEIP;
              FI;
    FI;rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
         IF OperandSize = 32
              THEN
                   IF stack not large enough for a 6-byte return address
                        THEN #SS(0); FI;
                   IF DEST[31:16] is not zero THEN #GP(0); FI;
                   Push(CS); (* Padded with 16 high-order bits *)
                   Push(EIP);
                   CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                   EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
              ELSE (* OperandSize = 16 *)
                   IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                   Push(CS);
                   Push(IP);
                   CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                   EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
         FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
         IF segment selector in target operand NULL
              THEN #GP(0); FI;
         IF segment selector index not within descriptor table limits
              THEN #GP(new code segment selector); FI;
         Read type and access rights of selected segment descriptor;
         IF IA32_EFER.LMA = 0
              THEN
                   IF segment type is not a conforming or nonconforming code segment, call
                   gate, task gate, or TSS
                        THEN #GP(segment selector); FI;
              ELSE
                   IF segment type is not a conforming or nonconforming code segment or
                   64-bit call gate,
                        THEN #GP(segment selector); FI;
         FI;
         Depending on type and access rights:
              GO TO CONFORMING-CODE-SEGMENT;
              GO TO NONCONFORMING-CODE-SEGMENT;
              GO TO CALL-GATE;
              GO TO TASK-GATE;
              GO TO TASK-STATE-SEGMENT;
FI;
```

```
CONFORMING-CODE-SEGMENT:
   IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
       THEN GP(new code segment selector); FI;
   IF DPL > CPL
       THEN #GP(new code segment selector); FI;
   IF segment not present
       THEN #NP(new code segment selector); FI;
   IF stack not large enough for return address
       THEN #SS(0); FI;
   tempEIP ← DEST(Offset);
   IF OperandSize = 16
       THEN
           tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
   IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
   segment limit)
       THEN #GP(0); FI;
   IF tempEIP is non-canonical
       THEN #GP(0); FI;
   IF OperandSize = 32
       THEN
           Push(CS); (* Padded with 16 high-order bits *)
           Push(EIP);
           CS ← DEST(CodeSegmentSelector);
           (* Segment descriptor information also loaded *)
           CS(RPL) ← CPL;
           EIP ← tempEIP;
       ELSE
           IF OperandSize = 16
               THEN
                   Push(CS);
                   Push(IP);
                   CS ← DEST(CodeSegmentSelector);
                   (* Segment descriptor information also loaded *)
                   CS(RPL) ← CPL;
                   EIP ← tempEIP;
               ELSE (* OperandSize = 64 *)
                   Push(CS); (* Padded with 48 high-order bits *)
                   Push(RIP);
                   CS ← DEST(CodeSegmentSelector);
                   (* Segment descriptor information also loaded *)
                   CS(RPL) ← CPL;
                   RIP ← tempEIP;
           FI;
   FI;
END;

NONCONFORMING-CODE-SEGMENT:
   IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
       THEN GP(new code segment selector); FI;
   IF (RPL > CPL) or (DPL ≠ CPL)
       THEN #GP(new code segment selector); FI;
   IF segment not present
       THEN #NP(new code segment selector); FI;
   IF stack not large enough for return address
```

```
            THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
            THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
            THEN #GP(0); FI;
    IF tempEIP is non-canonical
            THEN #GP(0); FI;
    IF OperandSize = 32
            THEN
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE
                IF OperandSize = 16
                    THEN
                        Push(CS);
                        Push(IP);
                        CS ← DEST(CodeSegmentSelector);
                        (* Segment descriptor information also loaded *)
                        CS(RPL) ← CPL;
                        EIP ← tempEIP;
                    ELSE (* OperandSize = 64 *)
                        Push(CS); (* Padded with 48 high-order bits *)
                        Push(RIP);
                        CS ← DEST(CodeSegmentSelector);
                        (* Segment descriptor information also loaded *)
                        CS(RPL) ← CPL;
                        RIP ← tempEIP;
                FI;
    FI;
END;

CALL-GATE:
    IF call gate (DPL < CPL) or (RPL > DPL)
            THEN #GP(call-gate selector); FI;
    IF call gate not present
            THEN #NP(call-gate selector); FI;
    IF call-gate code-segment selector is NULL
            THEN #GP(0); FI;
    IF call-gate code-segment selector index is outside descriptor table limits
            THEN #GP(call-gate code-segment selector); FI;
    Read call-gate code-segment descriptor;
    IF call-gate code-segment descriptor does not indicate a code segment
    or call-gate code-segment descriptor DPL > CPL
            THEN #GP(call-gate code-segment selector); FI;
    IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
            THEN #GP(call-gate code-segment selector); FI;
    IF call-gate code segment not present
```

```
        THEN #NP(call-gate code-segment selector); FI;
    IF call-gate code segment is non-conforming and DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit
        THEN
            TSSstackAddress ← (new code-segment DPL ∗ 8) + 4;
            IF (TSSstackAddress + 5) > current TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
            NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
        ELSE
            IF current TSS is 16-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL ∗ 4) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
                ELSE (* current TSS is 64-bit *)
                    TSSstackAddress ← (new code-segment DPL ∗ 8) + 4;
                    IF (TSSstackAddress + 7) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            FI;
    FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
    or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
    writable data segment)
        THEN #TS(NewSS); FI
    IF IA32_EFER.LMA = 0 and new stack segment not present
        THEN #SS(NewSS); FI;
    IF CallGateSize = 32
        THEN
            IF new stack does not have room for parameters plus 16 bytes
                THEN #SS(NewSS); FI;
            IF CallGate(InstructionPointer) not within new code-segment limit
                THEN #GP(0); FI;
            SS ← newSS; (* Segment descriptor information also loaded *)
            ESP ← newESP;
            CS:EIP ← CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            temp ← parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
```

```
        ELSE
            IF CallGateSize = 16
                THEN
                    IF new stack does not have room for parameters plus 8 bytes
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                        THEN #GP(0); FI;
                    SS ← newSS; (* Segment descriptor information also loaded *)
                    ESP ← newESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    temp ← parameter count from call gate, masked to 5 bits;
                    Push(parameters from calling procedure's stack, temp)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64 *)
                    IF pushing 32 bytes on the stack would use a non-canonical address
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) is non-canonical)
                        THEN #GP(0); FI;
                    SS ← NewSS; (* NewSS is NULL)
                    RSP ← NewESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
            FI;
    FI;
    CPL ← CodeSegment(DPL)
    CS(RPL) ← CPL
END;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
```

```
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP ← CallGate(CS:instruction pointer);
                (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
        FI;
    FI;
    CS(RPL) ← CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0) If the target offset in destination operand is beyond the new code segment limit.

If the segment selector in the destination operand is NULL.

If the code segment selector in the gate is NULL.

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

If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector) If a code segment or gate or TSS selector index is outside descriptor table limits.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.

If the segment selector from a call gate is beyond the descriptor table limits.

If the DPL for a code-segment obtained from a call gate is greater than the CPL.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.

If a memory operand effective address is outside the SS segment limit.

#SS(selector) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.

If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.

If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.

#NP(selector) If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.

#TS(selector) If the new stack segment selector and ESP are beyond the end of the TSS.

If the new stack segment selector is NULL.

If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.

If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.

If the new stack segment is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

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

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

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

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

If the target offset is beyond the code segment limit.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits.
	If code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
	If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CA CLAC	ZO	V/V	SMAP	Clear the AC flag in the EFLAGS register.

Instruction Operand Encoding

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

Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

Operation

```
EFLAGS.AC ← 0;
```

Flags Affected

AC cleared. Other flags are unaffected.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
NP 0F AE /7	CLFLUSH *m8*	M	Valid	Valid	Flushes cache line containing *m8*.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line.[1] They are not ordered with respect to executions of CLFLUSHOPT to different cache lines.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
CLFLUSH:    void _mm_clflush(void const *p)
```

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	RMI	V/V	SSE	Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.0F.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.0F.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Compare packed single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-4. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-4. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 0*
CMPLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 1*
CMPLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 2*
CMPUNORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 3*
CMPNEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 4*
CMPNLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 5*
CMPNLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 6*
CMPORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.

Table 3-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0*
VCMPLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1*
VCMPLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 2*
VCMPUNORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 3*
VCMPNEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 4*
VCMPNLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 5*
VCMPNLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 6*
VCMPORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 7*
VCMPEQ_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 8*
VCMPNGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 9*
VCMPNGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0AH*
VCMPFALSEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0CH*
VCMPGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0DH*
VCMPGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0EH*
VCMPTRUEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 10H*
VCMPLT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 11H*
VCMPLE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 12H*
VCMPUNORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 13H*
VCMPNEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 14H*
VCMPNLT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 15H*
VCMPNLE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 16H*
VCMPORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 17H*
VCMPEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 18H*
VCMPNGE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 19H*
VCMPNGT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1CH*
VCMPGE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1DH*
VCMPGT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1FH*

Operation

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

VCMPPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
         THEN
              IF (EVEX.b = 1) AND (SRC2 *is memory*)
                   THEN
                        CMP ← SRC1[i+31:i] OP5 SRC2[31:0]
                   ELSE
                        CMP ← SRC1[i+31:i] OP5 SRC2[i+31:i]
              FI;
              IF CMP = TRUE
                   THEN DEST[j] ← 1;
                   ELSE DEST[j] ← 0; FI;
         ELSE      DEST[j] ← 0                  ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VCMPPS (VEX.256 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
CMP4 ← SRC1[159:128] OP5 SRC2[159:128];
CMP5 ← SRC1[191:160] OP5 SRC2[191:160];
CMP6 ← SRC1[223:192] OP5 SRC2[223:192];
CMP7 ← SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ←000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ←000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 000000000H; FI;
DEST[MAX_VL-1:256] ← 0
```

VCMPPS (VEX.128 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[MAX_VL-1:128] ← 0
```

CMPPS (128-bit Legacy SSE version)

```
CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPS __mmask16 _mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

CMPSD—Compare Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	RMI	V/V	SSE2	Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	RVMI	V/V	AVX	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAX_VL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAX_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-6. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 0*
CMPLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 1*
CMPLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 2*
CMPUNORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 3*
CMPNEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 4*
CMPNLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 5*
CMPNLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 6*
CMPORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0*
VCMPLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 1*
VCMPLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 2*
VCMPUNORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 3*
VCMPNEQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 4*
VCMPNLTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 5*
VCMPNLESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 6*
VCMPORDSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 7*
VCMPEQ_UQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 8*
VCMPNGESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 9*
VCMPNGTSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0AH*
VCMPFALSESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0BH*
VCMPNEQ_OQSD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0CH*
VCMPGESD r*eg1, reg2, reg3*	VCMPSD r*eg1, reg2, reg3, 0DH*

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPGTSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 0EH*
VCMPTRUESD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 0FH*
VCMPEQ_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 10H*
VCMPLT_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 11H*
VCMPLE_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 12H*
VCMPUNORD_SSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 13H*
VCMPNEQ_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 14H*
VCMPNLT_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 15H*
VCMPNLE_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 16H*
VCMPORD_SSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 17H*
VCMPEQ_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 18H*
VCMPNGE_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 19H*
VCMPNGT_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1CH*
VCMPGE_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1DH*
VCMPGT_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1EH*
VCMPTRUE_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1FH*

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 ←EQ_OQ; OP5 ←EQ_OQ;
    1: OP3 ←LT_OS; OP5 ←LT_OS;
    2: OP3 ←LE_OS; OP5 ←LE_OS;
    3: OP3 ←UNORD_Q; OP5 ←UNORD_Q;
    4: OP3 ←NEQ_UQ; OP5 ←NEQ_UQ;
    5: OP3 ←NLT_US; OP5 ←NLT_US;
    6: OP3 ←NLE_US; OP5 ←NLE_US;
    7: OP3 ←ORD_Q; OP5 ←ORD_Q;
    8: OP5 ←EQ_UQ;
    9: OP5 ←NGE_US;
    10: OP5 ←NGT_US;
    11: OP5 ←FALSE_OQ;
    12: OP5 ←NEQ_OQ;
    13: OP5 ←GE_OS;
    14: OP5 ←GT_OS;
    15: OP5 ←TRUE_UQ;
    16: OP5 ←EQ_OS;
    17: OP5 ←LT_OQ;
    18: OP5 ←LE_OQ;
    19: OP5 ←UNORD_S;
    20: OP5 ←NEQ_US;
    21: OP5 ←NLT_UQ;
```

```
    22: OP5 ←NLE_UQ;
    23: OP5 ←ORD_S;
    24: OP5 ←EQ_US;
    25: OP5 ←NGE_UQ;
    26: OP5 ←NGT_UQ;
    27: OP5 ←FALSE_OS;
    28: OP5 ←NEQ_OS;
    29: OP5 ←GE_OQ;
    30: OP5 ←GT_OQ;
    31: OP5 ←TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSD (EVEX encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] ← 1;
                ELSE DEST[0] ← 0; FI;
    ELSE    DEST[0] ← 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

CMPSD (128-bit Legacy SSE version)

```
CMP0 ←DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[MAX_VL-1:64] (Unmodified)
```

VCMPSD (VEX.128 encoded version)

```
CMP0 ←SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSD __mmask8 _mm_cmp_sd_mask( __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_cmp_round_sd_mask( __m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 _mm_mask_cmp_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_mask_cmp_round_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1 Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

CMPSS—Compare Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	RMI	V/V	SSE	Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	RVMI	V/V	AVX	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

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

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAX_VL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAX_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either

by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-8. Pseudo-Op and CMPSS Implementation

Pseudo-Op	**CMPSS Implementation**
CMPEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 0*
CMPLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 1*
CMPLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 2*
CMPUNORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 3*
CMPNEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 4*
CMPNLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 5*
CMPNLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 6*
CMPORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-9.

Table 3-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	**CMPSS Implementation**
VCMPEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0*
VCMPLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 1*
VCMPLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 2*
VCMPUNORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 3*
VCMPNEQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 4*
VCMPNLTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 5*
VCMPNLESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 6*
VCMPORDSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 7*
VCMPEQ_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 8*
VCMPNGESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 9*
VCMPNGTSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0AH*
VCMPFALSESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0BH*
VCMPNEQ_OQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0CH*
VCMPGESS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 0DH*

Table 3-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPGTSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 0EH*
VCMPTRUESS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 10H*
VCMPLT_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 11H*
VCMPLE_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 12H*
VCMPUNORD_SSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 13H*
VCMPNEQ_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 14H*
VCMPNLT_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 15H*
VCMPNLE_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 16H*
VCMPORD_SSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 17H*
VCMPEQ_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 18H*
VCMPNGE_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 19H*
VCMPNGT_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1CH*
VCMPGE_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1DH*
VCMPGT_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1FH*

Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 ←EQ_OQ; OP5 ←EQ_OQ;
    1: OP3 ←LT_OS; OP5 ←LT_OS;
    2: OP3 ←LE_OS; OP5 ←LE_OS;
    3: OP3 ←UNORD_Q; OP5 ←UNORD_Q;
    4: OP3 ←NEQ_UQ; OP5 ←NEQ_UQ;
    5: OP3 ←NLT_US; OP5 ←NLT_US;
    6: OP3 ←NLE_US; OP5 ←NLE_US;
    7: OP3 ←ORD_Q; OP5 ←ORD_Q;
    8: OP5 ←EQ_UQ;
    9: OP5 ←NGE_US;
    10: OP5 ←NGT_US;
    11: OP5 ←FALSE_OQ;
    12: OP5 ←NEQ_OQ;
    13: OP5 ←GE_OS;
    14: OP5 ←GT_OS;
    15: OP5 ←TRUE_UQ;
    16: OP5 ←EQ_OS;
    17: OP5 ←LT_OQ;
    18: OP5 ←LE_OQ;
    19: OP5 ←UNORD_S;
    20: OP5 ←NEQ_US;
    21: OP5 ←NLT_UQ;
```

```
    22: OP5 ←NLE_UQ;
    23: OP5 ←ORD_S;
    24: OP5 ←EQ_US;
    25: OP5 ←NGE_UQ;
    26: OP5 ←NGT_UQ;
    27: OP5 ←FALSE_OS;
    28: OP5 ←NEQ_OS;
    29: OP5 ←GE_OQ;
    30: OP5 ←GT_OQ;
    31: OP5 ←TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSS (EVEX encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] ← 1;
                ELSE DEST[0] ← 0; FI;
    ELSE    DEST[0] ← 0                     ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

CMPSS (128-bit Legacy SSE version)

```
CMP0 ←DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] ←FFFFFFFFH;
ELSE DEST[31:0] ←00000000H; FI;
DEST[MAX_VL-1:32] (Unmodified)
```

VCMPSS (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSS __mmask8 _mm_cmp_ss_mask( __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_cmp_round_ss_mask( __m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 _mm_mask_cmp_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_round_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISD (all versions)

```
RESULT ←   OrderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ←0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2F /r COMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISS (all versions)

```
RESULT ←   OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CPUID—CPU Identification

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

Instruction Operand Encoding

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

Description

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

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

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

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

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

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

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

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

See also:

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

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

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

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"ineI"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value ∗ 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID.
	ECX	Feature Information (see Figure 3-7 and Table 3-10).
	EDX	Feature Information (see Figure 3-8 and Table 3-11).
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.
02H	EAX	Cache and TLB Information (see Table 3-12).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 214.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		Bits 07 - 05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache. Bits 13 - 10: Reserved. Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***. Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.
	EBX	Bits 11 - 00: L = System Coherency Line Size**. Bits 21 - 12: P = Physical Line partitions**. Bits 31 - 22: W = Ways of associativity**.
	ECX	Bits 31-00: S = Number of Sets**.
	EDX	Bit 00: Write-Back Invalidate/Invalidate. 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31 - 03: Reserved = 0. **NOTES:** * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** Add one to the return value to get the result. ***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache. **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15 - 00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	EBX	Bits 15 - 00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled. Bits 31 - 02: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT. Bits 07 - 04: Number of C1* sub C-states supported using MWAIT. Bits 11 - 08: Number of C2* sub C-states supported using MWAIT. Bits 15 - 12: Number of C3* sub C-states supported using MWAIT. Bits 19 - 16: Number of C4* sub C-states supported using MWAIT. Bits 23 - 20: Number of C5* sub C-states supported using MWAIT. Bits 27 - 24: Number of C6* sub C-states supported using MWAIT. Bits 31 - 28: Number of C7* sub C-states supported using MWAIT. **NOTE:** * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf	
06H	EAX	Bit 00: Digital temperature sensor is supported if set. Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved. Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bits 31 - 15: Reserved.
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31 - 04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0. Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 31 - 04: Reserved = 0.
	EDX	Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)
07H		Sub-leaf 0 (Input ECX = 0). *
	EAX	Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bits 17:16: Reserved. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bits 22 - 21: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bits 28 - 26: Reserved. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bits 31 - 30: Reserved.
	ECX	Bit 00: PREFETCHWT1. Bit 01: Reserved. Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bits 16 - 5: Reserved. Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID. Supports Read Processor ID if 1. Bits 29 - 23: Reserved. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved.
	EDX	Reserved. **NOTE:** * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Direct Cache Access Information Leaf	
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.
	EBX	Bit 00: Core cycle event not available if 1. Bit 01: Instruction retired event not available if 1. Bit 02: Reference cycles event not available if 1. Bit 03: Last-level cache reference event not available if 1. Bit 04: Last-level cache misses event not available if 1. Bit 05: Branch instruction retired event not available if 1. Bit 06: Branch mispredict retired event not available if 1. Bits 31 - 07: Reserved = 0.
	ECX	Reserved = 0.
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1). Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1). Reserved = 0.
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: Invalid. 1: SMT. 2: Core. 3-255: Reserved.
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).
	EAX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04 - 03: MPX state. Bits 07 - 05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 31 - 10: Reserved.
	EBX	Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
	EDX	Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
0DH	EAX	Bit 00: XSAVEOPT is available. Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set. Bit 02: Supports XGETBV with ECX = 1 if set. Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set. Bits 31 - 04: Reserved.
	EBX	Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCRO | IA32_XSS.
	ECX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07 - 00: Used for XCR0. Bit 08: PT state. Bit 09: Used for XCR0. Bits 31 - 10: Reserved.
	EDX	Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n ($0 \le n \le 31$) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n ($32 \le n \le 63$) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*.
	EBX	Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.
	ECX	Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31 - 02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)	
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved.
	L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)	
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31 - 03: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX. EAX Reserved. EBX Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31 - 04: Reserved. ECX Reserved. EDX Reserved.
	L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved. EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units. ECX Bits 01- 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved. EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
	L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved. EBX Bits 31 - 00: Bit-granular map of isolation/contention of allocation units. ECX Bits 31 - 00: Reserved. EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
	Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved. EBX Bits 31 - 00: Reserved. ECX Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	EDX Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
	Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)
12H	**NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bit 31 - 02: Reserved.
	EBX Bit 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.
	ECX Bit 31 - 00: Reserved.
	EDX Bit 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2^(EDX[7:0]). Bit 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2^(EDX[15:8]). Bits 31 - 16: Reserved.
	Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)
12H	**NOTES:** Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
	Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)
12H	**NOTES:** Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.
	EAX Bit 03 - 00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type 0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	Type 0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)
14H	**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.
	EBX Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 31 - 06: Reserved.
	ECX Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30 - 04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX Bits 31 - 00: Reserved.
	Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)
14H	EAX Bits 02 - 00: Number of configurable Address Ranges for filtering. Bits 15 - 03: Reserved. Bits 31 - 16: Bitmap of supported MTC period encodings.
	EBX Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings. Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.
	ECX Bits 31 - 00: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	EDX Bits 31 - 00: Reserved.
	Time Stamp Counter and Nominal Core Crystal Clock Information Leaf
15H	**NOTES:** If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. If ECX is 0, the nominal core crystal clock frequency is not enumerated. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.
	EAX Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX Bits 31 - 00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX Bits 31 - 00: Reserved = 0.
	Processor Frequency Information Leaf
16H	EAX Bits 15 - 00: Processor Base Frequency (in MHz). Bits 31 - 16: Reserved =0.
	EBX Bits 15 - 00: Maximum Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	ECX Bits 15 - 00: Bus (Reference) Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EDX Reserved.
	NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.
	System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)
17H	**NOTES:** Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.
	EAX Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.
	ECX Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)	
17H	EAX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
		NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.
	System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)	
17H		**NOTES:** Leaf 17H output depends on the initial value in ECX.
	EAX	Bits 31 - 00: Reserved = 0.
	EBX	Bits 31 - 00: Reserved = 0.
	ECX	Bits 31 - 00: Reserved = 0.
	EDX	Bits 31 - 00: Reserved = 0.
	Unimplemented CPUID Leaf Functions	
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode. Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET available in 64-bit mode. Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.
80000006H	EAX EBX	Reserved = 0. Reserved = 0.
	ECX	Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units.
	EDX	Reserved = 0.
		NOTES: * L2 associativity field encodings: 00H - Disabled. 01H - Direct mapped. 02H - 2-way. 04H - 4-way. 06H - 8-way. 08H - 16-way. 0FH - Fully associative.
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07 - 00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31 - 09: Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
80000008H	EAX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0.
	EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. **NOTES:** * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

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

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

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

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

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

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.

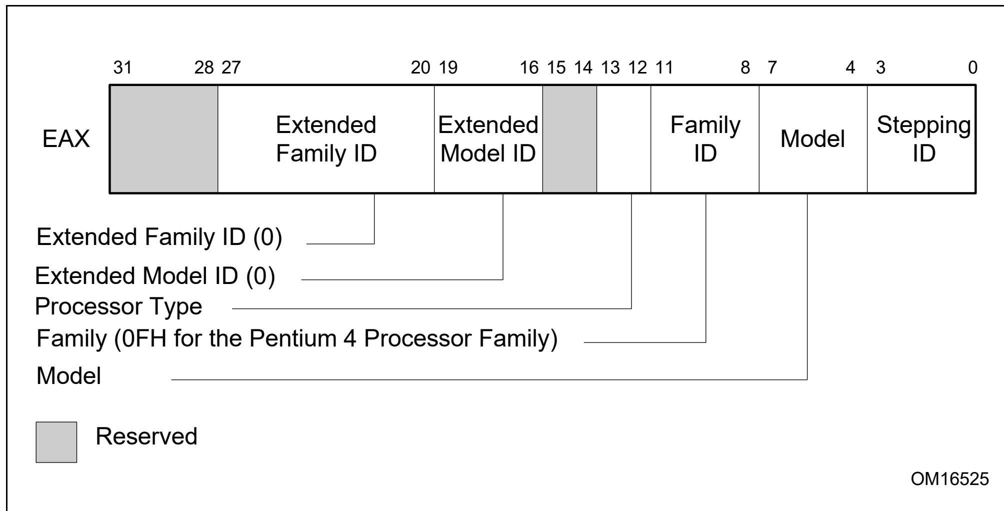


Figure 3-6. Version Information Returned by CPUID in EAX

Table 3-9. Processor Type Field

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

NOTE

See Chapter 19 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

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

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

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

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

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

NOTE

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

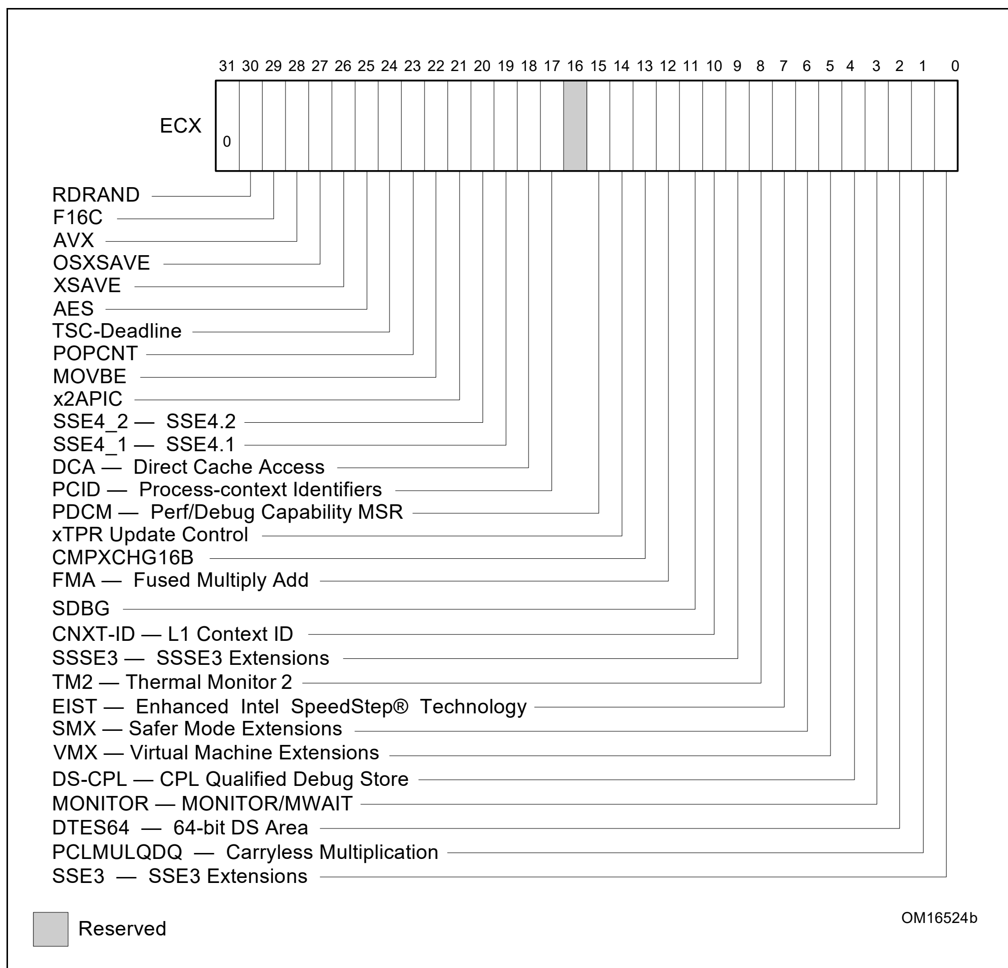


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	**PCLMULQDQ.** A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions.** A value of 1 indicates that the processor supports this technology.
6	SMX	**Safer Mode Extensions.** A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EIST	**Enhanced Intel SpeedStep® technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	**Perfmon and Debug Capability:** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.

Table 3-10. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

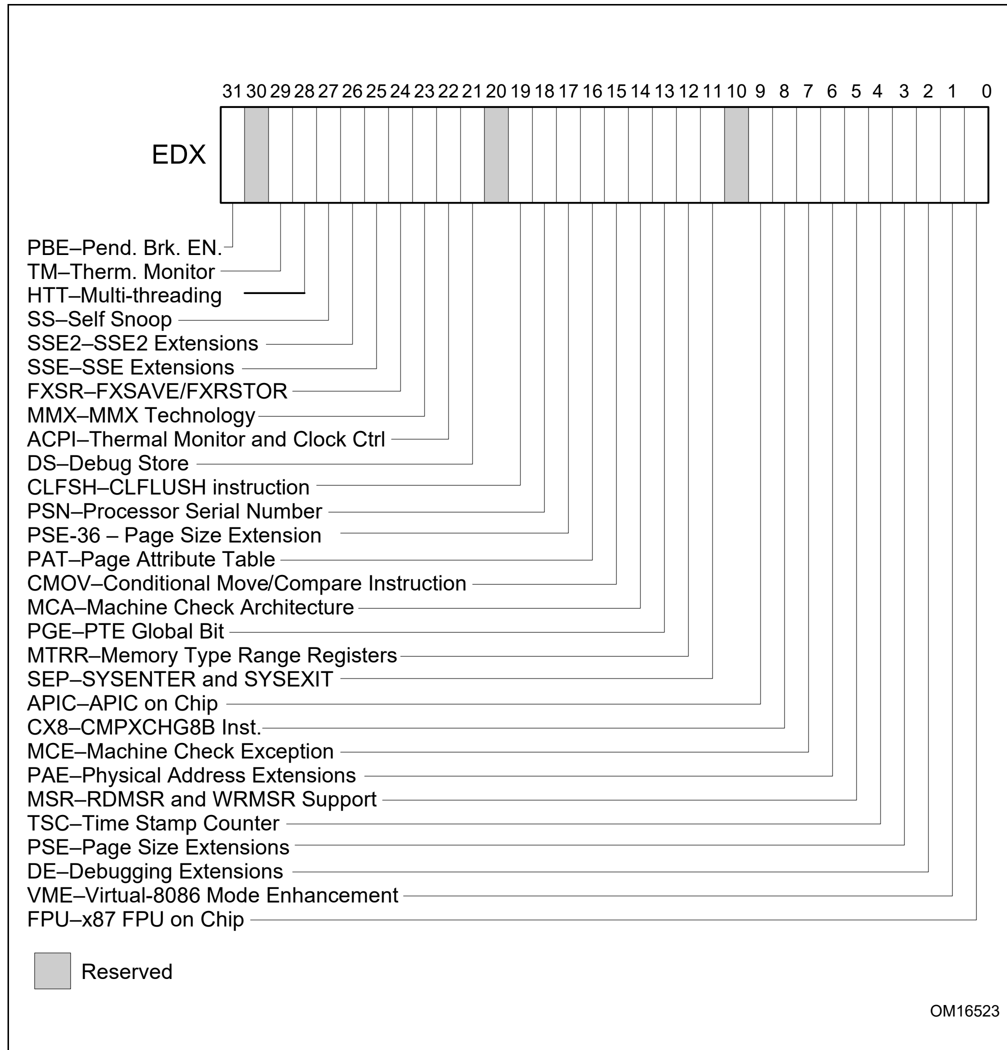


Figure 3-8. Feature Information Returned in the EDX Register

Table 3-11. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**Floating Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**Page Global Bit.** The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	**36-Bit Page Size Extension.** 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved

Table 3-11. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Max APIC IDs reserved field is Valid.** A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-12. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K-μop, 8-way set associative
71H	Cache	Trace cache: 16 K-μop, 8-way set associative
72H	Cache	Trace cache: 32 K-μop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GBbyte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

Example 3-1. Example of Cache and TLB Interpretation

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

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

Which means:

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

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

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

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

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

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
    IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
         Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
    FI;
```

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The Processor Brand String Method

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

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

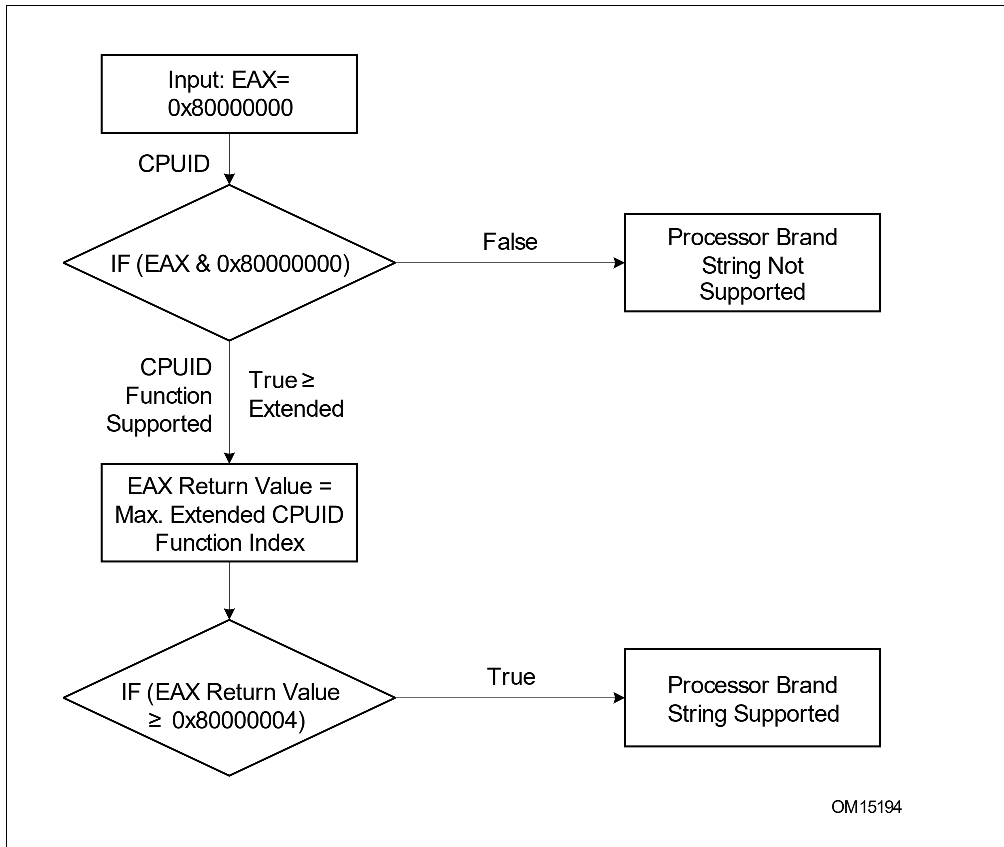


Figure 3-9. Determination of Support for the Processor Brand String

How Brand Strings Work

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

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-13. Processor Brand String Returned with Pentium 4 Processor

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

Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

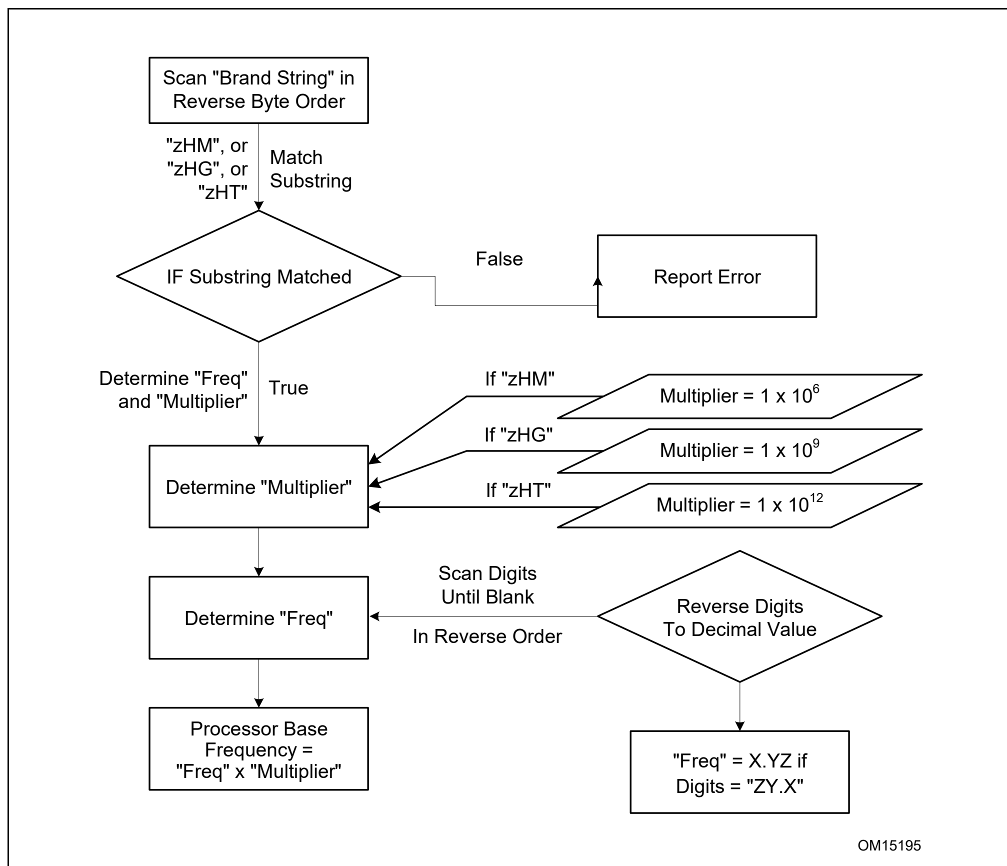


Figure 3-10. Algorithm for Extracting Processor Frequency

The Processor Brand Index Method

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

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

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

NOTES:

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

IA-32 Architecture Compatibility

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

Operation

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

CASE (EAX) OF
    EAX = 0:
        EAX ← Highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ← Initial APIC ID;
        ECX ← Feature flags; (* See Figure 3-7. *)
        EDX ← Feature flags; (* See Figure 3-8. *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-8. *)
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; (* See Table 3-8. *)
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    BREAK;
```

```
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 3-8. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H:
    EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Structured Extended Feature Flags Enumeration Leaf;
    ECX ← Structured Extended Feature Flags Enumeration Leaf;
    EDX ← Structured Extended Feature Flags Enumeration Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 3-8. *)
    EBX ← Direct Cache Access Information Leaf;
    ECX ← Direct Cache Access Information Leaf;
    EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-8. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
```

```
EAX = FH:
    EAX ← Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
    ECX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
    EDX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX ← Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Intel Resource Director Technology Allocation Enumeration Leaf;
    ECX ← Intel Resource Director Technology Allocation Enumeration Leaf;
    EDX ← Intel Resource Director Technology Allocation Enumeration Leaf;
BREAK;
    EAX = 12H:
    EAX ← Intel SGX Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Intel SGX Enumeration Leaf;
    ECX ← Intel SGX Enumeration Leaf;
    EDX ← Intel SGX Enumeration Leaf;
BREAK;
EAX = 14H:
    EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Intel Processor Trace Enumeration Leaf;
    ECX ← Intel Processor Trace Enumeration Leaf;
    EDX ← Intel Processor Trace Enumeration Leaf;
BREAK;
EAX = 15H:
    EAX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-8. *)
    EBX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
    ECX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
    EDX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
BREAK;
EAX = 16H:
    EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Processor Frequency Information Enumeration Leaf;
    ECX ← Processor Frequency Information Enumeration Leaf;
    EDX ← Processor Frequency Information Enumeration Leaf;
BREAK;
EAX = 17H:
    EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-8. *)
    EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;
    ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;
    EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;
BREAK;
EAX = 80000000H:
    EAX ← Highest extended function input value understood by CPUID;
    EBX ← Reserved;
    ECX ← Reserved;
    EDX ← Reserved;
BREAK;
EAX = 80000001H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← Extended Feature Bits (* See Table 3-8.*);
    EDX ← Extended Feature Bits (* See Table 3-8. *);
BREAK;
```

```
    EAX = 80000002H:
        EAX ← Processor Brand String;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Cache information;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000007H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = Misc Feature Flags;
    BREAK;
    EAX = 80000008H:
        EAX ← Reserved = Physical Address Size Information;
        EBX ← Reserved = Virtual Address Size Information;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX >= 40000000H and EAX <= 4FFFFFFFH:
    DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
        (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
        EAX ← Reserved; (* Information returned for highest basic information leaf. *)
        EBX ← Reserved; (* Information returned for highest basic information leaf. *)
        ECX ← Reserved; (* Information returned for highest basic information leaf. *)
        EDX ← Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

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

CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	RM	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	RM	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	RM	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512F	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	HV	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

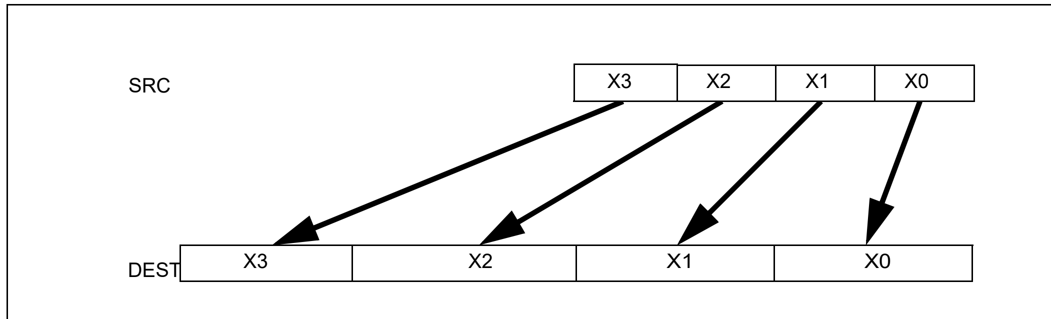


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTDQ2PD (VEX.256 encoded version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAX_VL-1:256] ← 0
```

VCVTDQ2PD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0
```

CVTDQ2PD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PD __m512d _mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d _mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d _mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src);
VCVTDQ2PD __m256d _mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d _mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)
```

Other Exceptions

VEX-encoded instructions, see Exceptions Type 5;

EVEX-encoded instructions, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDQ2PS xmm1, xmm2/m128	RM	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	RM	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	RM	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1with writemask k1.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. Bits (MAX_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper Bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);            ; refer to Table 2-4 in the Intel® Architecture Instruction Set Extensions Programming Reference
    ELSE
        SET_RM(MXCSR.RM);           ; refer to Table 2-4 in the Intel® Architecture Instruction Set Extensions Programming Reference
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTDQ2PS (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96)
DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224)
DEST[MAX_VL-1:256] ← 0
```

VCVTDQ2PS (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAX_VL-1:128] ← 0
```

CVTDQ2PS (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAX_VL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PS __m512 _mm512_cvtepi32_ps( __m512i a);
VCVTDQ2PS __m512 _mm512_mask_cvtepi32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_maskz_cvtepi32_ps( __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_cvt_roundepi32_ps( __m512i a, int r);
VCVTDQ2PS __m512 _mm512_mask_cvt_roundepi_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps( __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 _mm256_mask_cvtepi32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 _mm256_maskz_cvtepi32_ps( __mmask8 k, __m256i a);
VCVTDQ2PS __m128 _mm_mask_cvtepi32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 _mm_maskz_cvtepi32_ps( __mmask8 k, __m128i a);
CVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)
CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)
```

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2A /*r* CVTPI2PS *xmm, mm/m64*	RM	Valid	Valid	Convert two signed doubleword integers from *mm/m64* to two single-precision floating-point values in *xmm*.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS: __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5A /r CVTPS2PD xmm1, xmm2/m64	RM	V/V	SSE2	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	RM	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	RM	V/V	AVX	Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512F	Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL	Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	HV	V/V	AVX512F	Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

Converts two, four or eight packed single-precision floating-point values in the source operand (second operand) to two, four or eight packed double-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register. Bits (MAX_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

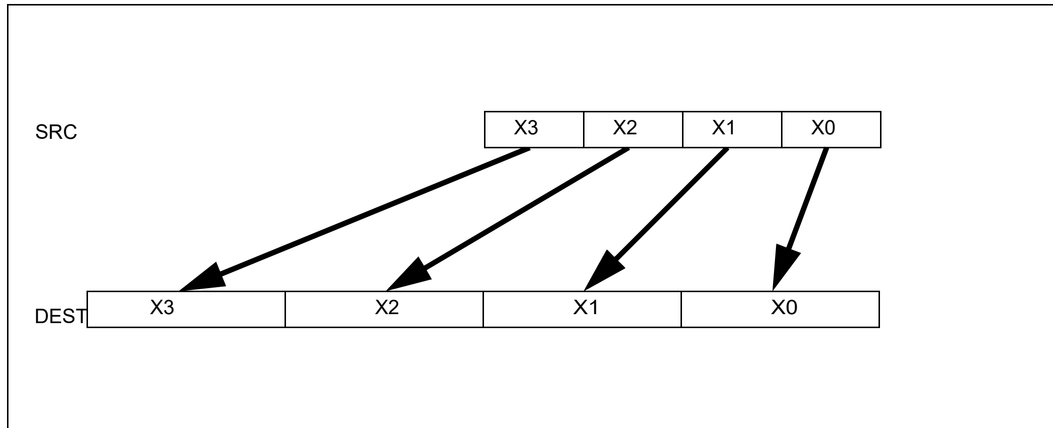


Figure 3-14. CVTPS2PD (VEX.256 encoded version)

Operation

VCVTPS2PD (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTPS2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
```

```
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VCVTPS2PD (VEX.256 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAX_VL-1:256] ← 0
```

VCVTPS2PD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0
```

CVTPS2PD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2PD __m512d _mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d _mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d _mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)
CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

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

Instruction Operand Encoding

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

Description

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

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI: __m64 _mm_cvtps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r [1] VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r [1] VCVTSD2SI r64, xmm1/m64	RM	V/N.E.[2]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	T1F	V/N.E.[2]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX encoded version)

```
IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE    DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI
```

(V)CVTSD2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD   If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	RM	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.
EVEX.NDS.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX_VL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VCVTSD2SS (VEX.128 encoded version)

```
DEST[31:0] ←Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

CVTSD2SS (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss( __mmask8 k, __m128 a,__m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss( __mmask8 k, __m128 a,__m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	RM	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	RM	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	RVM	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAX_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VCVTSI2SD (VEX.128 encoded version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

CVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvt_roundi32_sd(__m128d s, int a, int r);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)
```

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.

EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.

CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	RM	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	RM	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	RVM	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	T1S	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the "convert-from" source operand to a single-precision floating-point value in the destination operand (first operand). The "convert-from" source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. Bits (MAX_VL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VCVTSI2SS (VEX.128 encoded version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

CVTSI2SS (128-bit Legacy SSE version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);
```

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3NF.

CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	RM	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.
EVEX.NDS.LIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a single-precision floating-point value in the “convert-from” source operand to a double-precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX_VL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VCVTSS2SD (VEX.128 encoded version)
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0

CVTSS2SD (128-bit Legacy SSE version)
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAX_VL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a,__m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	RM	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r [1] VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r [1] VCVTSS2SI r64, xmm1/m32	RM	V/N.E.[2]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	T1F	V/N.E.[2]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^{w-1}, where w represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX encoded version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

(V)CVTSS2SI (Legacy and VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int _mm_cvtss_i32(__m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32(__m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64(__m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2C /r CVTTPS2PI *mm, xmm/m64*	RM	Valid	Valid	Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* using truncation.

Instruction Operand Encoding

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

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r [1] VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r [1] VCVTTSD2SI r64, xmm1/m64	T1F	V/N.E.[2]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	T1F	V/N.E.[2]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2SI int _mm_cvttsd_i32( __m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32( __m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64( __m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64( __m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32( __m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64( __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	RM	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r [1] VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r [1] VCVTTSS2SI r64, xmm1/m32	RM	V/N.E.[2]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	T1F	V/N.E.[2]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSS2SI int _mm_cvttss_i32( __m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32( __m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64( __m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64( __m128 a, int sae);
CVTTSS2SI int _mm_cvttss_si32( __m128 a);
CVTTSS2SI __int64 _mm_cvttss_si64( __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	RM	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.
VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.
VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.
EVEX.NDS.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VDIVPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VDIVPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[159:128] ←SRC1[159:128] / SRC2[159:128]
DEST[191:160]←SRC1[191:160] / SRC2[191:160]
DEST[223:192] ←SRC1[223:192] / SRC2[223:192]
DEST[255:224] ←SRC1[255:224] / SRC2[255:224].
DEST[MAX_VL-1:256] ←0;
```

VDIVPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[MAX_VL-1:128] ←0
```

DIVPS (128-bit Legacy SSE version)

DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m512 _mm512_div_ps(__m512 a, __m512 b);
VDIVPS __m512 _mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 _mm512_div_round_ps(__m512 a, __m512 b, int);
VDIVPS __m512 _mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);
DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

DIVSD—Divide Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	RM	V/V	SSE2	Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64.
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.
EVEX.NDS.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VDIVSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] / SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

DIVSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] / SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd( __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd( __m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	RM	V/V	SSE	Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.
EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VDIVSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

DIVSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] / SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 77	EMMS	ZO	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

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

Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
x87FPUTagWord ← FFFFH;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_empty()
```

Flags Affected

None

Protected Mode Exceptions

#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /1 FXRSTOR *m512byte*	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*.
NP REX.W + 0F AE /1 FXRSTOR64 *m512byte*	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*.

Instruction Operand Encoding

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

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-43 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-43, Table 3-46, or Table 3-47. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

Operation

```
IF 64-Bit Mode
    THEN
        (x87 FPU, MMX, XMM15-XMM0, MXCSR)    Load(SRC);
    ELSE
         (x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);
FI;
```

x87 FPU and SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /0 FXSAVE *m512byte*	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*.
NP REX.W + 0F AE /0 FXSAVE64 *m512byte*	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*.

Instruction Operand Encoding

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

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

The operation of FXSAVE in non-64-bit modes is described first.

Non-64-Bit Mode Operation

Table 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Rsvd	FCS	FIP[31:0]		FOP	Rsvd	FTW	FSW	FCW	**0**
MXCSR_MASK		MXCSR		Rsrvd	FDS		FDP[31:0]		**16**
Reserved			ST0/MM0						**32**
Reserved			ST1/MM1						**48**
Reserved			ST2/MM2						**64**
Reserved			ST3/MM3						**80**
Reserved			ST4/MM4						**96**
Reserved			ST5/MM5						**112**
Reserved			ST6/MM6						**128**
Reserved			ST7/MM7						**144**
XMM0									**160**
XMM1									**176**
XMM2									**192**
XMM3									**208**
XMM4									**224**
XMM5									**240**
XMM6									**256**
XMM7									**272**
Reserved									**288**

Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved									304
Reserved									320
Reserved									336
Reserved									352
Reserved									368
Reserved									384
Reserved									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-43 are defined in Table 3-44.

Table 3-44. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU opcode field.
FIP	x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU instruction pointer.

Table 3-44. Field Definitions (Contd.)

Field	Definition
FCS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
FDP	x87 FPU Instruction Operand (Data) Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit DP offset. 64-bit mode without REX.W — 32-bit DP offset. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the x87 FPU operand pointer.
FDS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_ MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See “Guidelines for Writing to the MXCSR Register” in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

```
R7 R6 R5 R4 R3 R2 R1 R0
11 xx xx xx 11 11 11 11
```

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

```
R7 R6 R5 R4 R3 R2 R1 R0
0  1  1  1  0  0  0  0
```

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be

used by an application program to pass a "clean" x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.

- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-45.

Table 3-45. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above.				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-46), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.

Table 3-46. Layout of the 64-bit-mode FXSAVE64 Map (requires REX.W = 1)

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

Table 3-47. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	FCS	FIP[31:0]		FOP	Reserved	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Reserved	FDS		FDP[31:0]		16
Reserved			ST0/MM0						32
Reserved			ST1/MM1						48
Reserved			ST2/MM2						64
Reserved			ST3/MM3						80
Reserved			ST4/MM4						96
Reserved			ST5/MM5						112
Reserved			ST6/MM6						128
Reserved			ST7/MM7						144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

Operation

```
IF 64-Bit Mode
    THEN
        IF REX.W = 1
            THEN
                DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
                MXCSR);
            ELSE
                DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
        FI;
    ELSE
        DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
FI;
```

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

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

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	ZO	Valid	Valid	Interrupt 3—trap to debugger.
CD *ib*	INT *imm8*	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	ZO	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm8	NA	NA	NA

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the "normal" 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

Table 3-51. Decision Table

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

– Don't Care.

Y Yes, action taken.

Blank Action not taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

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

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

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

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

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (VM = 1 and IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                IF (IA32_EFER.LMA = 0)
                    THEN (* Protected mode, or virtual-8086 mode interrupt *)
                        GOTO PROTECTED-MODE;
                ELSE (* IA-32e mode interrupt *)
                    GOTO IA-32e-MODE;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number « 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS ← IDT(Descriptor (vector_number « 2), selector));
    EIP ← IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF INTO and CS.L = 1 (64-bit mode)
        THEN #UD;
```

```
    FI;
    IF ((vector_number « 4) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
        IF local/global bit is set to local or index not within GDT limits
            THEN #GP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Access TSS descriptor in GDT;
        IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF TSS not present
            THEN #NP(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(EXT); FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
```

```
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            IF new code segment is conforming or new code-segment DPL = CPL
                THEN
                    GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                    #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
            FI;
    FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
        (* Identify stack-segment selector for new privilege level in current TSS *)
            IF current TSS is 32-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL « 3) + 4;
                    IF (TSSstackAddress + 5) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                    NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
                ELSE    (* current TSS is 16-bit *)
                    TSSstackAddress ← (new code-segment DPL « 2) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
            FI;
            IF NewSS is NULL
                THEN #TS(EXT); FI;
            IF NewSS index is not within its descriptor-table limits
            or NewSS RPL ≠ new code-segment DPL
```

```
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            Read new stack-segment descriptor for NewSS in GDT or LDT;
            IF new stack-segment DPL ≠ new code-segment DPL
            or new stack-segment Type does not indicate writable data segment
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF NewSS is not present
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IA-32e mode *)
            IF IDT-gate IST = 0
                THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
                ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
            FI;
            IF (TSSstackAddress + 7) > current TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
    FI;
    IF IDT gate is 32-bit
            THEN
                IF new stack does not have room for 24 bytes (error code pushed)
                or 20 bytes (no error code pushed)
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            FI
        ELSE
            IF IDT gate is 16-bit
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limits
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            ESP ← NewESP;
            SS ← NewSS; (* Segment descriptor information also loaded *)
        ELSE (* IA-32e mode *)
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
            SS ← NewSS;
    FI;
    IF IDT gate is 32-bit
        THEN
```

```
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit IDT gate *)
                    CS:RIP ← Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    IF IDT gate is 32-bit
            THEN
                Push(far pointer to old stack);
                (* Old SS and ESP, 3 words padded to 4 *)
                Push(EFLAGS);
                Push(far pointer to return instruction);
                (* Old CS and EIP, 3 words padded to 4 *)
                Push(ErrorCode); (* If needed, 4 bytes *)
            ELSE
                IF IDT gate 16-bit
                    THEN
                        Push(far pointer to old stack);
                        (* Old SS and SP, 2 words *)
                        Push(EFLAGS(15-0]);
                        Push(far pointer to return instruction);
                        (* Old CS and IP, 2 words *)
                        Push(ErrorCode); (* If needed, 2 bytes *)
                    ELSE (* 64-bit IDT gate *)
                        Push(far pointer to old stack);
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8-bytes *)
            FI;
    FI;
    CPL ← new code-segment DPL;
    CS(RPL) ← CPL;
    IF IDT gate is interrupt gate
        THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Identify stack-segment selector for privilege level 0 in current TSS *)
    IF current TSS is 32-bit
        THEN
            IF TSS limit < 9
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 8);
```

```
            NewESP ← 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 4);
            NewESP ← 2 bytes loaded from (current TSS base + 2);
FI;
IF NewSS is NULL
    THEN #TS(EXT); FI; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 40 bytes (error code pushed)
        or 36 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IDT gate is 16-bit)
        IF new stack does not have room for 20 bytes (error code pushed)
        or 18 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
```

```
    Push(TempEFlags);
    Push(CS);
    Push(EIP);
    GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
    FS ← 0;
    DS ← 0;
    ES ← 0;
    CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
    IF OperandSize = 32
        THEN
            EIP ← Gate(instruction pointer);
        ELSE (* OperandSize is 16 *)
            EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, DPL = CPL or conforming segment *)
    IF IA32_EFER.LMA = 1 (* IA-32e mode *)
        IF IDT-descriptor IST ≠ 0
            THEN
                TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
                ELSE NewRSP ← RSP;
                FI;
    FI;
    IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
            or 12 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(EXT); (* Error code contains NULL selector *)
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limit
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
    FI;
    IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
        THEN
```

```
                Push (EFLAGS);
                Push (far pointer to return instruction); (* 3 words padded to 4 *)
                CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
                Push (ErrorCode); (* If any *)
            ELSE
                IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
                    THEN
                        Push (FLAGS);
                        Push (far pointer to return location); (* 2 words *)
                        CS:IP ← Gate(CS:IP);
                        (* Segment descriptor information also loaded *)
                        Push (ErrorCode); (* If any *)
                    ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                        Push(far pointer to old stack);
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8 bytes *)
                        CS:RIP ← GATE(CS:RIP);
                        (* Segment descriptor information also loaded *)
                FI;
        FI;
        CS(RPL) ← CPL;
        IF IDT gate is interrupt gate
            THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
        TF ← 0;
        NT ← 0;
        VM ← 0;
        RF ← 0;
END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(error_code) If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.

If the segment selector in the interrupt-, trap-, or task gate is NULL.

If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.

If the vector selects a descriptor outside the IDT limits.

If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.

If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.

If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(error_code)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT *n,* INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.

#TS(error_code) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.

If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.

If the stack segment selector in the TSS is NULL.

If the stack segment for the TSS is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

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

#BP If the INT 3 instruction is executed.

#OF If the INTO instruction is executed and the OF flag is set.

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code) If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.

If the segment selector in the 64-bit interrupt or trap gate is NULL.

If the vector selects a descriptor outside the IDT limits.

If the vector points to a gate which is in non-canonical space.

If the vector points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.

If the descriptor pointed to by the gate selector is outside the descriptor table limit.

If the descriptor pointed to by the gate selector is in non-canonical space.

If the descriptor pointed to by the gate selector is not a code segment.

If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.

If the descriptor pointed to by the gate selector has DPL > CPL.

#SS(error_code) If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.

If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).

#NP(error_code) If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.

#TS(error_code) If an attempt to load RSP from the TSS causes an access to non-canonical space.

If the RSP from the TSS is outside descriptor table limits.

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

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

IRET/IRETD—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	ZO	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	ZO	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	ZO	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

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

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, "Handling Multiple NMIs" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

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

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
    FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ← Pop();
                (* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE
```

```
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
        ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS ← tempEFLAGS;
    ESP ← Pop();
    SS ← Pop(); (* Pop 2 words; throw away high-order word *)
    ES ← Pop(); (* Pop 2 words; throw away high-order word *)
    DS ← Pop(); (* Pop 2 words; throw away high-order word *)
    FS ← Pop(); (* Pop 2 words; throw away high-order word *)
    GS ← Pop(); (* Pop 2 words; throw away high-order word *)
    CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
```

```
                ESP ← Pop();
                SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE IF OperandSize = 16
            THEN
                ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                SS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RSP ← Pop();
                SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
    FI;
    IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                        THEN #GP(0); FI;
    FI;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32 or or OperandSize = 64
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
    IF CPL ≤ IOPL
            THEN EFLAGS(IF) ← tempEFLAGS; FI;
    IF CPL = 0
            THEN
                EFLAGS(IOPL) ← tempEFLAGS;
                IF OperandSize = 32 or OperandSize = 64
                    THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
    CPL ← CS(RPL);
    FOR each SegReg in (ES, FS, GS, and DS)
            DO
                tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
                IF tempDesc(DPL) < CPL AND tempDesc(Type) is data or non-conforming code
                    THEN (* Segment register invalid *)
                        SegReg ← NULL;
                FI;
            OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                        THEN #GP(0); FI;
    FI;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
    IF CPL ≤ IOPL
            THEN EFLAGS(IF) ← tempEFLAGS; FI;
```

```
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
END;

IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
    ELSE IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop();
            tempEFLAGS ← Pop();
        ELSE IF OperandSize = 16
            THEN
                EIP ← Pop(); (* 16-bit pop; clear upper bits *)
                CS ← Pop(); (* 16-bit pop *)
                tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    RIP ← Pop();
                    CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    tempRFLAGS ← Pop();
    FI;
    IF CS.RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE
            IF instruction began in 64-Bit Mode
                THEN
                    IF OperandSize = 32
                        THEN
                            ESP ← Pop();
                            SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                            SS ← Pop(); (* 16-bit pop *)
                        ELSE (* OperandSize = 64 *)
                            RSP ← Pop();
                            SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    FI;
            FI;
            GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is less than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.
	If the return code segment selector is NULL.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return instruction pointer is not within the return code segment limit.
	If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

LAR—Load Access Rights Byte

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 02 /*r*	LAR *r16, r16/m16*	RM	Valid	Valid	*r16* ← access rights referenced by *r16/m16*
0F 02 /*r*	LAR *reg, r32/m16*[1]	RM	Valid	Valid	*reg* ← access rights referenced by *r32/m16*

NOTES:

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

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

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-52.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-52. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```
IF Offset(SRC) > descriptor table limit
    THEN
        ZF ← 0;
    ELSE
        SegmentDescriptor ← descriptor referenced by SRC;
        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or SegmentDescriptor(Type) is not valid for instruction
            THEN
                ZF ← 0;
            ELSE
                DEST ← access rights from SegmentDescriptor as given in Description section;
                ZF ← 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LDMXCSR—Load MXCSR Register

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

Instruction Operand Encoding

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

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See "MXCSR Control and Status Register" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the MXCSR register and its contents.

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

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

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

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

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

Operation

```
MXCSR ← m32;
```

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

Numeric Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

LFENCE—Load Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE E8	LFENCE	ZO	Valid	Valid	Serializes load operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete **before** the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

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

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

Operation

```
Wait_On_Following_Instructions_Until(preceding_instructions_complete);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_lfence(void)
```

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT *m16&32*	M	N.E.	Valid	Load *m* into GDTR.
0F 01 /3	LIDT *m16&32*	M	N.E.	Valid	Load *m* into IDTR.
0F 01 /2	LGDT *m16&64*	M	Valid	N.E.	Load *m* into GDTR.
0F 01 /3	LIDT *m16&64*	M	Valid	N.E.	Load *m* into IDTR.

Instruction Operand Encoding

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

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See "SGDT—Store Global Descriptor Table Register" in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

Operation

```
IF Instruction is LIDT
    THEN
        IF OperandSize = 16
            THEN
                IDTR(Limit) ← SRC[0:15];
                IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:79];
                FI;
        FI;
    ELSE (* Instruction is LGDT *)
        IF OperandSize = 16
            THEN
                GDTR(Limit) ← SRC[0:15];
                GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    GDTR(Limit) ← SRC[0:15];
                    GDTR(Base) ← SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    GDTR(Limit) ← SRC[0:15];
                    GDTR(Base) ← SRC[16:79];
                FI;
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If the current privilege level is not 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

LMSW—Load Machine Status Word

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /6	LMSW *r/m*16	M	Valid	Valid	Loads *r/m*16 in machine status word of CR0.

Instruction Operand Encoding

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

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
CR0[0:3] ← SRC[0:3];
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The LMSW instruction is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

5. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, M-U.

Change to this chapter: Updates to the following instructions: MASKMOVQ, MAXPS, MAXSD, MAXSS, MFENCE, MINPS, MINSD, MINSS, MONITOR, MOVAPS, MOVD/MOVQ, MOVHLPS, MOVHPS, MOVLHPS, MOVLPS, MOVMSKPS, MOVNTI, MOVNTPS, MOVNTQ, MOVQ, MOVUPS, MULPS, MULSD, MULSS, NOP, ORPS, PABSB/PABSW/PABSD, PACKSSDW/PACKSSWB, PACKUSWB, PADDB/PADDW/PADDD/PADDQ, PADDSB/PADDSW, PADDUSB/PADDUSW, PALIGNR, PAND, PANDN, PAVGB/PAVGW, PCMPEQB/PCMPEQW/PCMPEQD, PCMPGTB/PCMPGTW/PCMPGTD, PEXTRB/PEXTRD/PEXTRQ, PEXTRW, PHADDD/PHADDW, PHADDSW, PHSUBD/PHSUBW, PHSUBSW, PINSRB/PINSRD/PINSRQ, PINSRW, PMADDUBSW, PMADDWD, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHRSW, PMULHUW, PMULHW, PMULLW, PMULUDQ, POR, PSADBW, PSHUFB, PSHUFW, PSIGNB/PSIGND/PSIGNW, PSLLW/PSLLD/PSLLQ, PSRAD/PSRAW, PSRLD/PSRLW/PSRLQ, PSUBB/PSUBD/PSUBW, PSUBQ, PSUBSB/PSUBSW, PSUBUSB/PSUBUSW, PUNPCKHBW/PUNPCKHDQ/PUNPCKHWD, PUNPCKLBW/PUNPCKLDQ/PUNPCKLWD, PXOR, RCPPS, RDPID, RDPKRU, RDPMC, RET, RSQRTPS, SFENCE, SGDT, SHA1MSG1, SHA1MSG2, SHA1NEXTE, SHA1RNDS4, SHA256MSG1, SHA256MSG2, SHA256RNDS2, SHUFPS, SIDT, SQRTPS, SQRTSD, SQRTSS, STAC, STMXCSR, SUBPS, SUBSD, SUBSS, UCOMISD, UCOMISS, UNPCKHPS and UNPCKLPS.

MASKMOVQ—Store Selected Bytes of Quadword

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

Instruction Operand Encoding

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

Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

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

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

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	RM	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

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

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

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMAXPS (VEX.256 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ←MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ←MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] ←MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] ←MAX(SRC1[255:224], SRC2[255:224])
DEST[MAX_VL-1:256] ←0
```

VMAXPS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[MAX_VL-1:128] ←0
```

MAXPS (128-bit Legacy SSE version)

DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ←MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ←MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ←MAX(DEST[127:96], SRC[127:96])
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPS __m512 _mm512_max_ps(__m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps(__mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps(__m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps(__mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps(__mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	RM	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

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

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

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

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VMAXSD (VEX.128 encoded version)

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

MAXSD (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	RM	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

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

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

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

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

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
         ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
         ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
         ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
         ELSE DEST ←SRC2;
    FI;
}
```

VMAXSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
    ELSE
         IF *merging-masking*                    ; merging-masking
              THEN *DEST[31:0] remains unchanged*
              ELSE                               ; zeroing-masking
                   THEN DEST[31:0] ← 0
         FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VMAXSS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

MAXSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSS __m128 _mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MFENCE—Memory Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F0	MFENCE	ZO	Valid	Valid	Serializes load and store operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.[1] The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_mfence(void)
```

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	RM	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

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

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

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMINPS (VEX.256 encoded version)

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

VMINPS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAX_VL-1:128] ←0
```

MINPS (128-bit Legacy SSE version)

DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMINPS __m512 _mm512_min_ps(__m512 a, __m512 b);
VMINPS __m512 _mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_min_round_ps(__m512 a, __m512 b, int);
VMINPS __m512 _mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 _mm512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 _mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 _mm256_maskz_min_ps(__mmask8 k, __m256 a, __m25 b);
VMINPS __m128 _mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 _mm_maskz_min_ps(__mmask8 k, __m128 a, __m128 b);
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	RM	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

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

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

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

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

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

MINSD (VEX.128 encoded version)

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

MINSD (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	RM	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

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

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

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

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

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VMINSS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C8	MONITOR	ZO	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.

Instruction Operand Encoding

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

Description

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

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

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

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, "Multiple-Processor Management" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If ECX ≠ 0.
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
	If ECX ≠ 0.
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.
	If RCX ≠ 0.
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

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

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	RM	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	MR	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	RM	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	MR	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	RM	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	MR	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

(E)VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMOVAPS (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMOVAPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-1:256] ← 0
```

VMOVAPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPS (VEX.128 encoded version, load - and register copy)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-1:128] ← 0
```

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

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

(V)MOVAPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVAPS __m512 _mm512_load_ps( void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps( __mmask16 k, void * m);
VMOVAPS void _mm512_store_ps( void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps( void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps(__m256 a, __mmask8 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps( void * d, __mmask8 k, __m256 a);
VMOVAPS __m128 _mm_mask_load_ps(__m128 a, __mmask8 k, void * s);
VMOVAPS __m128 _mm_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm_mask_store_ps( void * d, __mmask8 k, __m128 a);
MOVAPS __m256 _mm256_load_ps (float * p);
MOVAPS void _mm256_store_ps(float * p, __m256 a);
MOVAPS __m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /*r* MOVD *mm, r/m32*	RM	V/V	MMX	Move doubleword from *r/m32* to *mm*.
NP REX.W + 0F 6E /*r* MOVQ *mm, r/m64*	RM	V/N.E.	MMX	Move quadword from *r/m64* to *mm*.
NP 0F 7E /*r* MOVD *r/m32, mm*	MR	V/V	MMX	Move doubleword from *mm* to *r/m32*.
NP REX.W + 0F 7E /*r* MOVQ *r/m64, mm*	MR	V/N.E.	MMX	Move quadword from *mm* to *r/m64*.
66 0F 6E /*r* MOVD *xmm, r/m32*	RM	V/V	SSE2	Move doubleword from *r/m32* to *xmm*.
66 REX.W 0F 6E /*r* MOVQ *xmm, r/m64*	RM	V/N.E.	SSE2	Move quadword from *r/m64* to *xmm*.
66 0F 7E /*r* MOVD *r/m32, xmm*	MR	V/V	SSE2	Move doubleword from *xmm* register to *r/m32*.
66 REX.W 0F 7E /*r* MOVQ *r/m64, xmm*	MR	V/N.E.	SSE2	Move quadword from *xmm* register to *r/m64*.
VEX.128.66.0F.W0 6E / VMOVD *xmm1, r32/m32*	RM	V/V	AVX	Move doubleword from *r/m32* to *xmm1*.
VEX.128.66.0F.W1 6E /r VMOVQ *xmm1, r64/m64*	RM	V/N.E[1].	AVX	Move quadword from *r/m64* to *xmm1*.
VEX.128.66.0F.W0 7E /r VMOVD *r32/m32, xmm1*	MR	V/V	AVX	Move doubleword from *xmm1* register to *r/m32*.
VEX.128.66.0F.W1 7E /r VMOVQ *r64/m64, xmm1*	MR	V/N.E[1].	AVX	Move quadword from *xmm1* register to *r/m64*.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	T1S-RM	V/V	AVX512F	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	T1S-RM	V/N.E.[1]	AVX512F	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	T1S-MR	V/V	AVX512F	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	T1S-MR	V/N.E.[1]	AVX512F	Move quadword from xmm1 register to r/m64.

NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

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

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

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAX_VL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (when destination operand is MMX technology register)

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

MOVD (when destination operand is XMM register)

```
DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[VLMAX-1:128] (Unmodified)
```

MOVD (when source operand is MMX technology or XMM register)

```
DEST ← SRC[31:0];
```

VMOVD (VEX-encoded version when destination is an XMM register)

```
DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] ← 0
```

MOVQ (when destination operand is XMM register)

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[VLMAX-1:128] (Unmodified)
```

MOVQ (when destination operand is r/m64)

```
DEST[63:0] ← SRC[63:0];
```

MOVQ (when source operand is XMM register or r/m64)

```
DEST ← SRC[63:0];
```

VMOVQ (VEX-encoded version when destination is an XMM register)

```
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0
```

VMOVD (EVEX-encoded version when destination is an XMM register)

```
DEST[31:0] ← SRC[31:0]
DEST[511:32] ← 0H
```

VMOVQ (EVEX-encoded version when destination is an XMM register)

```
DEST[63:0] ← SRC[63:0]
DEST[511:64] ← 0H
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVD:	__m64 _mm_cvtsi32_si64 (int i)
MOVD:	int _mm_cvtsi64_si32 (__m64m)
MOVD:	__m128i _mm_cvtsi32_si128 (int a)
MOVD:	int _mm_cvtsi128_si32 (__m128i a)
MOVQ:	__int64 _mm_cvtsi128_si64(__m128i);
MOVQ:	__m128i _mm_cvtsi64_si128(__int64);
VMOVD	__m128i _mm_cvtsi32_si128(int);
VMOVD	int _mm_cvtsi128_si32(__m128i);
VMOVQ	__m128i _mm_cvtsi64_si128 (__int64);
VMOVQ	__int64 _mm_cvtsi128_si64(__m128i);
VMOVQ	__m128i _mm_loadl_epi64(__m128i * s);
VMOVQ	void _mm_storel_epi64(__m128i * d, __m128i s);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

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

MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit two-argument form)

```
DEST[63:0] ← SRC[127:64]
DEST[MAX_VL-1:64] (Unmodified)
```

VMOVHLPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	T2	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	MR	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	MR	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	T2-MR	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T2-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAX_VL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE load)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAX_VL-1:128] (Unmodified)
```

VMOVHPS (VEX.128 and EVEX encoded load)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] ← 0
```

VMOVHPS (store)

```
DEST[63:0] ← SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVHPS __m128 _mm_loadh_pi ( __m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAX_VL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit two-argument form)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAX_VL-1:128] (Unmodified)
```

VMOVLHPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.NDS.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	T2	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	MR	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	MR	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	T2-MR	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T2-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAX_VL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

VMOVLPS (VEX.128 & EVEX encoded load)

DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

VMOVLPS (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 50 /r MOVMSKPS *reg, xmm*	RM	V/V	SSE	Extract 4-bit sign mask from *xmm* and store in *reg*. The upper bits of *r32* or *r64* are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS *reg, xmm2*	RM	V/V	AVX	Extract 4-bit sign mask from *xmm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS *reg, ymm2*	RM	V/V	AVX	Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of *r32* or *r64* are zeroed.

Instruction Operand Encoding[1]

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

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

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

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

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

Operation

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
    THEN DEST[31:4] ← ZeroExtend;
    ELSE DEST[63:4] ← ZeroExtend;
FI;
```

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

VMOVMSKPS (VEX.256 encoded version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)

int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C3 /r	MOVNTI *m32, r32*	MR	Valid	Valid	Move doubleword from *r32* to *m32* using non-temporal hint.
NP REX.W + 0F C3 /r	MOVNTI *m64, r64*	MR	Valid	N.E.	Move quadword from *r64* to *m64* using non-temporal hint.

Instruction Operand Encoding

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

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI: void _mm_stream_si32 (int *p, int a)

MOVNTI: void _mm_stream_si64(__int64 *p, __int64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	MR	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	MR	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	MR	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	FVM	V/V	AVX512VL AVX512F	Move packed single-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	FVM	V/V	AVX512VL AVX512F	Move packed single-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	FVM	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

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

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VMOVNTPS (EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAX_VL-1:VL] ← 0
```

1. ModRM.MOD = 011B required

MOVNTPS

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);

MOVNTPS void _mm_stream_ps (float * p, __m128d a);

VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTQ—Store of Quadword Using Non-Temporal Hint

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

Instruction Operand Encoding

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

Description

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

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVNTQ:     void _mm_stream_pi(__m64 * p, __m64 a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ *mm, mm/m64*	RM	V/V	MMX	Move quadword from *mm/m64* to *mm*.
NP 0F 7F /r MOVQ *mm/m64, mm*	MR	V/V	MMX	Move quadword from *mm* to *mm/m64*.
F3 0F 7E /r MOVQ *xmm1, xmm2/m64*	RM	V/V	SSE2	Move quadword from *xmm2/mem64* to *xmm1*.
VEX.128.F3.0F.WIG 7E /r VMOVQ *xmm1, xmm2/m64*	RM	V/V	AVX	Move quadword from *xmm2* to *xmm1*.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	T1S-RM	V/V	AVX512F	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ *xmm2/m64, xmm1*	MR	V/V	SSE2	Move quadword from *xmm1* to *xmm2/mem64*.
VEX.128.66.0F.WIG D6 /r VMOVQ *xmm1/m64, xmm2*	MR	V/V	AVX	Move quadword from *xmm2* register to *xmm1/m64*.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	T1S-MR	V/V	AVX512F	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

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

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

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

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

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

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

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination:
DEST[63:0] ← SRC[63:0]
DEST[MAX_VL-1:64] ← 0

VMOVQ (7E) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with memory source:
DEST[63:0] ← SRC[63:0]
DEST[:MAX_VL-1:64] ← 0

VMOVQ (D6) with memory dest:
DEST[63:0] ← SRC2[63:0]

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ __m128i _mm_loadu_si64(void * s);
VMOVQ void _mm_storeu_si64(void * d, __m128i s);

MOVQ m128i _mm_mov_epi64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	RM	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	MR	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	RM	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.0F 11.WIG /r VMOVUPS xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F 10.WIG /r VMOVUPS ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.0F 11.WIG /r VMOVUPS ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAX_VL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

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

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAX_VL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMOVUPS (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]← SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
         THEN DEST[i+31:i] ← SRC[i+31:i]
         ELSE
              IF *merging-masking*                  ; merging-masking
                   THEN *DEST[i+31:i] remains unchanged*
                   ELSE  DEST[i+31:i] ← 0           ; zeroing-masking
              FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMOVUPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAX_VL-1:256] ← 0
```

VMOVUPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVUPS (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAX_VL-1:128] ← 0
```

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

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

(V)MOVUPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPS __m512 _mm512_loadu_ps( void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps( __mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps( void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps( void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps( void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps( void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps ( float * p);
MOVUPS void _mm256 _storeu_ps( float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps ( float * p);
MOVUPS void _mm_storeu_ps( float *p, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	RM	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAX_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VMULPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[159:128] ←SRC1[159:128] * SRC2[159:128]
DEST[191:160]←SRC1[191:160] * SRC2[191:160]
DEST[223:192] ←SRC1[223:192] * SRC2[223:192]
DEST[255:224] ←SRC1[255:224] * SRC2[255:224].
DEST[MAX_VL-1:256] ←0;
```

VMULPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] ←0
```

MULPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MULSD—Multiply Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	RM	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.NDS.LIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	T1S	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

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

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
    ELSE
        IF *merging-masking*            ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                        ; zeroing-masking
                THEN DEST[63:0] ← 0
            FI
    FI;
ENDFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VMULSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

MULSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] * SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	RM	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.
EVEX.NDS.LIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	T1S	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

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

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
            FI
    FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VMULSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

MULSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] * SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 90	NOP	ZO	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

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

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

VORPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VORPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAX_VL-1:256] ← 0
```

VORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAX_VL-1:128] ← 0
```

ORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PABSB/PABSW/PABSD/PABSQ — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB *mm1, mm2/m64*	RM	V/V	SSSE3	Compute the absolute value of bytes in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1C /r PABSB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1D /r[1] PABSW *mm1, mm2/m64*	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1D /r PABSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1E /r[1] PABSD *mm1, mm2/m64*	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1E /r PABSD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1C /r VPABSB *xmm1, xmm2/m128*	RM	V/V	AVX	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1D /r VPABSW *xmm1, xmm2/m128*	RM	V/V	AVX	Compute the absolute value of 16- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1E /r VPABSD *xmm1, xmm2/m128*	RM	V/V	AVX	Compute the absolute value of 32- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.256.66.0F38.WIG 1C /r VPABSB *ymm1, ymm2/m256*	RM	V/V	AVX2	Compute the absolute value of bytes in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1D /r VPABSW *ymm1, ymm2/m256*	RM	V/V	AVX2	Compute the absolute value of 16-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1E /r VPABSD *ymm1, ymm2/m256*	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	FVM	V/V	AVX512BW	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.

EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	FVM	V/V	AVX512BW	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PABSB with 128 bit operands:

```
    Unsigned DEST[7:0] ←ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] ←ABS(SRC[127:120])
```

VPABSB with 128 bit operands:

```
    Unsigned DEST[7:0] ←ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120]←ABS(SRC[127:120])
```

VPABSB with 256 bit operands:

```
    Unsigned DEST[7:0]←ABS(SRC[7: 0])
    Repeat operation for 2nd through 31st bytes
    Unsigned DEST[255:248]←ABS(SRC[255:248])
```

VPABSB (EVEX encoded versions)

```
    (KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PABSW with 128 bit operands:

```
    Unsigned DEST[15:0]←ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112]←ABS(SRC[127:112])
```

VPABSW with 128 bit operands:

```
    Unsigned DEST[15:0] ←ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112]←ABS(SRC[127:112])
```

VPABSW with 256 bit operands:

```
    Unsigned DEST[15:0]←ABS(SRC[15:0])
    Repeat operation for 2nd through 15th 16-bit words
    Unsigned DEST[255:240] ←ABS(SRC[255:240])
```

VPABSW (EVEX encoded versions)

```
    (KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+15:i] ← ABS(SRC[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PABSD with 128 bit operands:

```
    Unsigned DEST[31:0]←ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96]←ABS(SRC[127:96])
```

VPABSD with 128 bit operands:

```
    Unsigned DEST[31:0]←ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96]←ABS(SRC[127:96])
```

VPABSD with 256 bit operands:

```
    Unsigned DEST[31:0] ←ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] ←ABS(SRC[255:224])
```

VPABSD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+31:i] ← ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] ← ABS(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPABSQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+63:i] ← ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD  __m512i _mm512_abs_epi32( __m512i a);
VPABSD  __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64( __m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64( __mmask8 k, __m128i a);
PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)
```

VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPABSD/Q, see Exceptions Type E4.

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 63 /r[1] PACKSSWB *mm1, mm2/m64*	RM	V/V	MMX	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /r PACKSSWB *xmm1, xmm2/m128*	RM	V/V	SSE2	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation.
NP 0F 6B /r[1] PACKSSDW *mm1, mm2/m64*	RM	V/V	MMX	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /r PACKSSDW *xmm1, xmm2/m128*	RM	V/V	SSE2	Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Converts 8 packed signed word integers from *xmm2* and from *xmm3/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Converts 4 packed signed doubleword integers from *xmm2* and from *xmm3/m128* into 8 packed signed word integers in *xmm1* using signed saturation.
VEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Converts 16 packed signed word integers from *ymm2* and from *ymm3/m256* into 32 packed signed byte integers in *ymm1* using signed saturation.
VEX.NDS.256.66.0F.WIG 6B /r VPACKSSDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Converts 8 packed signed doubleword integers from *ymm2* and from *ymm3/m256* into 16 packed signed word integers in *ymm1*using signed saturation.
EVEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *xmm2* and from *xmm3/m128* into packed signed byte integers in *xmm1* using signed saturation under writemask k1.
EVEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *ymm2* and from *ymm3/m256* into packed signed byte integers in *ymm1* using signed saturation under writemask k1.
EVEX.NDS.512.66.0F.WIG 63 /r VPACKSSWB *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Converts packed signed word integers from *zmm2* and from *zmm3/m512* into packed signed byte integers in *zmm1* using signed saturation under writemask k1.
EVEX.NDS.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *xmm2* and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

EVEX.NDS.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *ymm2* and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.NDS.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512BW	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

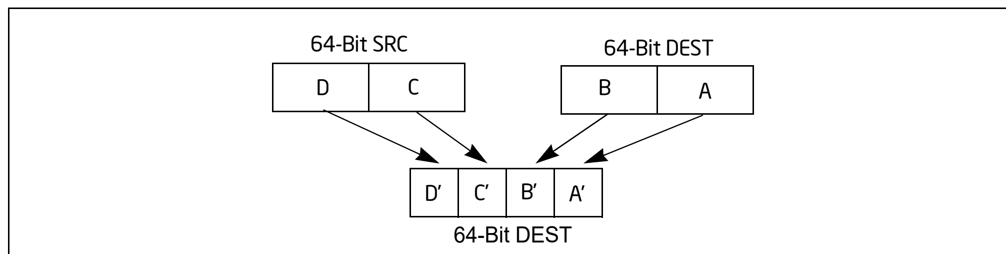


Figure 4-6. Operation of the PACKSSDW Instruction Using 64-bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e. greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAX_VL-1:128] (Unmodified)
```

PACKSSDW instruction (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAX_VL-1:128] (Unmodified)
```

VPACKSSWB instruction (VEX.128 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[MAX_VL-1:128] ← 0;
```

VPACKSSDW instruction (VEX.128 encoded version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[MAX_VL-1:128] ← 0;
```

VPACKSSWB instruction (VEX.256 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAX_VL-1:256] ← 0;
```

VPACKSSDW instruction (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAX_VL-1:256] ← 0;
```

VPACKSSWB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
```

```
    TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
```

```
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPACKSSDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);
```

```
    TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*        ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32( __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16( __mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 67 /r[1] PACKUSWB *mm, mm/m64*	RM	V/V	MMX	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /r PACKUSWB *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Converts 8 signed word integers from *xmm2* and 8 signed word integers from *xmm3/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Converts 16 signed word integers from *ymm2* and 16signed word integers from *ymm3/m256* into 32 unsigned byte integers in *ymm1* using unsigned saturation.
EVEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1{k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Converts signed word integers from *xmm2* and signed word integers from *xmm3/m128* into unsigned byte integers in *xmm1* using unsigned saturation under writemask k1.
EVEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1{k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Converts signed word integers from *ymm2* and signed word integers from *ymm3/m256* into unsigned byte integers in *ymm1* using unsigned saturation under writemask k1.
EVEX.NDS.512.66.0F.WIG 67 /r VPACKUSWB *zmm1{k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Converts signed word integers from *zmm2* and signed word integers from *zmm3/m512* into unsigned byte integers in *zmm1* using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

PACKUSWB (with 64-bit operands)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (Legacy SSE instruction)

```
DEST[7:0]←SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 encoded version)

```
DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

```
    DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[VLMAX-1:128] ← 0;
```

VPACKUSWB (VEX.256 encoded version)

```
    DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] ←SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] ←SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    DEST[143:136] ←SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    DEST[151:144] ←SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    DEST[159:152] ←SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
```

VPACKUSWB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

```
TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);
```

```
    TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSWB__m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB__m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);

PACKUSWB: __m64 _mm_packs_pu16(__m64 m1, __m64 m2)

(V)PACKUSWB: __m128i _mm_packus_epi16(__m128i m1, __m128i m2)

VPACKUSWB: __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FC /r[1] PADDB *mm, mm/m64*	RM	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
NP 0F FD /r[1] PADDW *mm, mm/m64*	RM	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
NP 0F FE /r[1] PADDD *mm, mm/m64*	RM	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
NP 0F D4 /r[1] PADDQ *mm, mm/m64*	RM	V/V	MMX	Add packed quadword integers from *mm/m64* and *mm*.
66 0F FC /r PADDB *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed word integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W0 FE /r VPADDD *xmm1 {k1}{z}, xmm2,* xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed doubleword integers from *xmm2, and xmm3/m128/m32bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W1 D4 /r VPADDQ *xmm1 {k1}{z}, xmm2,* xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Add packed quadword integers from *xmm2, and xmm3/m128/m64bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed word integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F.W0 FE /r VPADDD *ymm1 {k1}{z}, ymm2,* ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed doubleword integers from *ymm2, ymm3/m256/m32bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ *ymm1 {k1}{z}, ymm2,* ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Add packed quadword integers from *ymm2, ymm3/m256/m64bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FC /r VPADDB *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed byte integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FD /r VPADDW *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed word integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	FV	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ *zmm1 {k1}{z}, zmm2,* zmm3/m512/m64bcst	FV	V/V	AVX512F	Add packed quadword integers from *zmm2, zmm3/m512/m64bcst* and store in *zmm1* using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too

large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAX_VL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

PADDW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

PADDD (with 64-bit operands)

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

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE instruction)

```
DEST[7:0]← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← DEST[127:120] + SRC[127:120];
DEST[MAX_VL-1:128] (Unmodified)
```

PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← DEST[127:112] + SRC[127:112];
DEST[MAX_VL-1:128] (Unmodified)
```

PADDD (Legacy SSE instruction)

```
DEST[31:0]← DEST[31:0]  + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96]← DEST[127:96] + SRC[127:96];
DEST[MAX_VL-1:128] (Unmodified)
```

PADDQ (Legacy SSE instruction)

```
DEST[63:0]← DEST[63:0]  + SRC[63:0];
DEST[127:64]← DEST[127:64] + SRC[127:64];
DEST[MAX_VL-1:128] (Unmodified)
```

VPADDB (VEX.128 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← SRC1[127:120] + SRC2[127:120];
DEST[MAX_VL-1:128] ← 0;
```

VPADDW (VEX.128 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← SRC1[127:112] + SRC2[127:112];
DEST[MAX_VL-1:128] ← 0;
```

VPADDD (VEX.128 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← SRC1[127:96] + SRC2[127:96];
DEST[MAX_VL-1:128] ← 0;
```

VPADDQ (VEX.128 encoded instruction)

```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[MAX_VL-1:128] ← 0;
```

VPADDB (VEX.256 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248]← SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240]← SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] ← SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 encoded instruction)

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

VPADDB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPADDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPADDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPADDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPADDB__m512i _mm512_add_epi8 ( __m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 ( __m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32( __m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32( __mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32( __mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32( __mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64( __m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64( __mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64( __mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64( __mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b );
PADDW __m128i _mm_add_epi16 ( __m128i a, __m128i b);
PADDD  __m128i _mm_add_epi32 ( __m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 ( __m128i a, __m128i b);
```

```
VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b );
VPADDW __m256i _mm256_add_epi16 ( __m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 ( __m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 ( __m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_pi64(__m64 m1, __m64 m2)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDD/Q, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB *mm, mm/m64*	RM	V/V	MMX	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /r PADDSB *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F ED /r[1] PADDSW *mm, mm/m64*	RM	V/V	MMX	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /r PADDSW *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed signed byte integers from *xmm3/m128* and *xmm2* saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed signed word integers from *xmm3/m128* and *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed signed byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EC /r VPADDSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed signed byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed signed word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG ED /r VPADDSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed signed word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

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

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

PADDSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (with 128-bit operands)

```
DEST[7:0] ←SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPADDSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PADDSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 7th words *)
    DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

PADDSW (with 128-bit operands)

```
    DEST[15:0]  ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[VLMAX-1:128] ← 0
```

VPADDSW (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB: __m64 _mm_adds_pi8(__m64 m1, __m64 m2)

(V)PADDSB: __m128i _mm_adds_epi8 (__m128i a, __m128i b)

VPADDSB: __m256i _mm256_adds_epi8 (__m256i a, __m256i b)

PADDSW: __m64 _mm_adds_pi16(__m64 m1, __m64 m2)

(V)PADDSW: __m128i _mm_adds_epi16 (__m128i a, __m128i b)

VPADDSW: __m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB__m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW__m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB__m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB__m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB__m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB__m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB__m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB__m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DC /*r*[1] PADDUSB *mm, mm/m64*	RM	V/V	MMX	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /*r* PADDUSB *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F DD /*r*[1] PADDUSW *mm, mm/m64*	RM	V/V	MMX	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /*r* PADDUSW *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.
VEX.NDS.128.660F.WIG DC /r VPADDUSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed unsigned byte integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed unsigned word integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed unsigned byte integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed unsigned word integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.NDS.128.66.0F.WIG DC /r VPADDUSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG DC /r VPADDUSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed unsigned byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.

EVEX.NDS.512.66.0F.WIG DD /r VPADDUSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Add packed unsigned word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

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

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

PADDUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPADDUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

VPADDUSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPADDUSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] = 0
                FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PADDUSB:      __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW:      __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB:   __m128i _mm_adds_epu8 ( __m128i a, __m128i b)
(V)PADDUSW:   __m128i _mm_adds_epu16 ( __m128i a, __m128i b)
VPADDUSB:     __m256i _mm256_adds_epu8 ( __m256i a, __m256i b)
VPADDUSW:     __m256i _mm256_adds_epu16 ( __m256i a, __m256i b)
VPADDUSB__m512i _mm512_adds_epu8 ( __m512i a, __m512i b)
VPADDUSW__m512i _mm512_adds_epu16 ( __m512i a, __m512i b)
VPADDUSB__m512i _mm512_mask_adds_epu8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_mask_adds_epu16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A 0F /r ib[1] PALIGNR *mm1, mm2/m64, imm8*	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *mm1*.
66 0F 3A 0F /r ib PALIGNR *xmm1, xmm2/m128, imm8*	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *xmm1*.
VEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Concatenate *xmm2* and *xmm3/m128*, extract byte aligned result shifted to the right by constant value in *imm8* and result is stored in *xmm1*.
VEX.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and *ymm3/m256* into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in *imm8* from each intermediate result, and two 16-byte results are stored in *ymm1*.
EVEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	FVMI	V/V	AVX512VL AVX512BW	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	FVMI	V/V	AVX512VL AVX512BW	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.NDS.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	FVMI	V/V	AVX512BW	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
FVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX,

XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

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

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

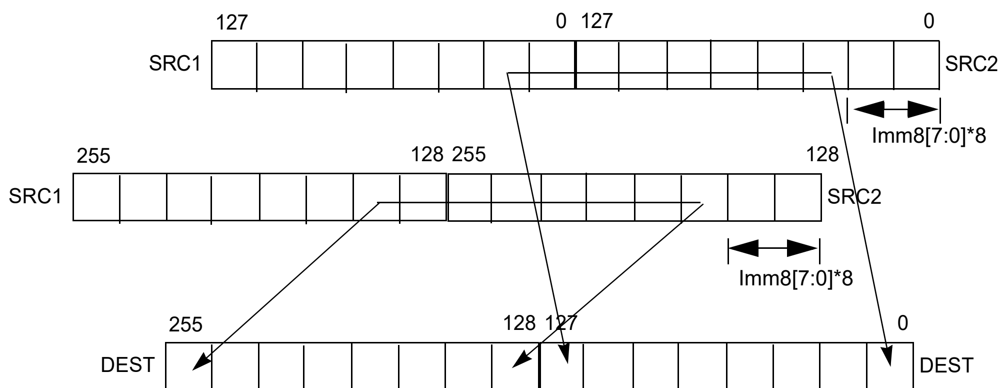


Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (with 128-bit operands)

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] (Unmodified)
```

VPALIGNR (VEX.128 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] ← 0
```

VPALIGNR (VEX.256 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAX_VL-1:128] ← temp1[127:0]
```

VPALIGNR (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR l ← 0 TO VL-1 with increments of 128
    temp1[255:0] ← ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] ← temp1[127:0]
ENDFOR;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR: __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)

(V)PALIGNR: __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)

VPALIGNR: __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)

VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)

VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)

VPALIGNR __m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)

VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)

VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)

VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)

VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /r[1] PAND *mm, mm/m64*	RM	V/V	MMX	Bitwise AND *mm/m64* and *mm*.
66 0F DB /*r* PAND *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise AND of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DB /r VPAND *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise AND of *xmm3/m128* and *xmm*.
VEX.NDS.256.66.0F.WIG DB /r VPAND *ymm1, ymm2, ymm3/.m256*	RVM	V/V	AVX2	Bitwise AND of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.NDS.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DB /*r* VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DB /*r* VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit operand)

```
DEST ← DEST AND SRC
```

PAND (128-bit Legacy SSE version)

```
DEST ← DEST AND SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPAND (VEX.128 encoded version)

```
DEST ← SRC1 AND SRC2
DEST[VLMAX-1:128] ← 0
```

VPAND (VEX.256 encoded instruction)

```
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])
DEST[VLMAX-1:256] ← 0
```

VPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
         THEN
              IF (EVEX.b = 1) AND (SRC2 *is memory*)
                   THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
                   ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
              FI;
         ELSE
              IF *merging-masking*                      ; merging-masking
                   THEN *DEST[i+63:i] remains unchanged*
                   ELSE                                 ; zeroing-masking
                        DEST[i+63:i] ← 0
              FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64( __m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64( __mmask8 k, __m128i a, __m128i b);
PAND:   __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND:__m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND:      __m256i _mm256_and_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DF /r[1] PANDN *mm, mm/m64*	RM	V/V	MMX	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /r PANDN *xmm1, xmm2/m128*	RM	V/V	SSE2	Bitwise AND NOT of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DF /r VPANDN *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise AND NOT of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG DF /r VPANDN *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise AND NOT of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.NDS.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PANDN (64-bit operand)

```
DEST ← NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE version)

```
DEST ← NOT(DEST) AND SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPANDN (VEX.128 encoded version)

```
DEST ← NOT(SRC1) AND SRC2
DEST[VLMAX-1:128] ← 0
```

VPANDN (VEX.256 encoded instruction)

```
DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[VLMAX-1:256] ← 0
```

VPANDND (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[31:0])
                ELSE DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPANDNQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64( __m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64( __mmask8 k, __m128i a, __m128i b);
PANDN: __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN:__m128i _mm_andnot_si128 ( __m128i a, __m128i b)
VPANDN:        __m256i _mm256_andnot_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E0 /r[1] PAVGB *mm1, mm2/m64*	RM	V/V	SSE	Average packed unsigned byte integers from *mm2/m64* and *mm1* with rounding.
66 0F E0, /r PAVGB *xmm1, xmm2/m128*	RM	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
NP 0F E3 /r[1] PAVGW *mm1, mm2/m64*	RM	V/V	SSE	Average packed unsigned word integers from mm2/m64 and *mm1* with rounding.
66 0F E3 /r PAVGW *xmm1, xmm2/m128*	RM	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Average packed unsigned byte integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Average packed unsigned word integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to *ymm1*.
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Average packed unsigned word integers from *ymm2, ymm3/m256* with rounding to *ymm1*.
EVEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1* {k1}{z}, *xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from xmm2, and *xmm3/m128* with rounding and store to xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1* {k1}{z}, *ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E0 /r VPAVGB *zmm1* {k1}{z}, *zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Average packed unsigned byte integers from zmm2, and *zmm3/m512* with rounding and store to zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1* {k1}{z}, *xmm2, xmm3/m128*	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from xmm2, *xmm3/m128* with rounding to xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1* {k1}{z}, *ymm2, ymm3/m256*	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from ymm2, *ymm3/m256* with rounding to ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E3 /r VPAVGW *zmm1* {k1}{z}, *zmm2, zmm3/m512*	FVM	V/V	AVX512BW	Average packed unsigned word integers from zmm2, *zmm3/m512* with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding register destination are zeroed.

Operation

PAVGB (with 64-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 6 *)
DEST[63:56] ← (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (with 64-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 and 3 *)
DEST[63:48] ← (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (with 128-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (with 128-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 encoded version)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

VPAVGW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

VPAVGB (VEX.256 encoded instruction)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31)
DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 encoded instruction)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15)
DEST[255:14]) ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
```

VPAVGB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPAVGW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                        ; (* Temp sum before shifting is 17 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB: __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW: __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB: __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW: __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 74 /r[1] PCMPEQB *mm, mm/m64*	RM	V/V	MMX	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /r PCMPEQB *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed bytes in *xmm2/m128* and *xmm1* for equality.
NP 0F 75 /r[1] PCMPEQW *mm, mm/m64*	RM	V/V	MMX	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /r PCMPEQW *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed words in *xmm2/m128* and *xmm1* for equality.
NP 0F 76 /r[1] PCMPEQD *mm, mm/m64*	RM	V/V	MMX	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /r PCMPEQD *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed doublewords in *xmm2/m128* and *xmm1* for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed bytes in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed words in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed doublewords in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW *ymm1, ymm2, ymm3 /m256*	RVM	V/V	AVX2	Compare packed words in *ymm3/m256* and *ymm2* for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD *ymm1, ymm2, ymm3 /m256*	RVM	V/V	AVX2	Compare packed doublewords in *ymm3/m256* and *ymm2* for equality.
EVEX.NDS.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	FVM	V/V	AVX512VL AVX512BW	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	FVM	V/V	AVX512VL AVX512BW	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	FVM	V/V	AVX512BW	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	FVM	V/V	AVX512VL AVX512BW	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	FVM	V/V	AVX512VL AVX512BW	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	FVM	V/V	AVX512BW	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

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

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (with 64-bit operands)

```
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] = SRC[63:56]
        THEN DEST[63:56] ← FFH;
        ELSE DEST[63:56] ← 0; FI;
```

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPEQB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 encoded version)

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

VPCMPEQB (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPEQB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] == SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQW (with 64-bit operands)

```
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] = SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPEQW (with 128-bit operands)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPEQW (VEX.128 encoded version)

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

VPCMPEQW (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPEQW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] == SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0               ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQD (with 64-bit operands)

```
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] = SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; FI;
```

PCMPEQD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 encoded version)

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

VPCMPEQD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPEQD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0               ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB: __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW: __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD: __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB: __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW: __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD: __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB: __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW: __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD: __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQD, see Exceptions Type E4.
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB *mm, mm/m64*	RM	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /r PCMPGTB *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 65 /r[1] PCMPGTW *mm, mm/m64*	RM	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /r PCMPGTW *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 66 /r[1] PCMPGTD *mm, mm/m64*	RM	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /r PCMPGTD *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed word integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed doubleword integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed word integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed doubleword integers in *ymm2* and *ymm3/m256* for greater than.
EVEX.NDS.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

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

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (with 64-bit operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] ← FFH;
        ELSE DEST[63:56] ← 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPGTB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 encoded version)

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

VPCMPGTB (VEX.256 encoded version)
```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPGTB (EVEX encoded versions)
```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTW (with 64-bit operands)
```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW (with 128-bit operands)
```
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 encoded version)
```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPGTW (VEX.256 encoded version)
```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPGTW (EVEX encoded versions)
```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
```

```
            ELSE        DEST[j] ← 0                         ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTD (with 64-bit operands)

```
    IF DEST[31:0] > SRC[31:0]
            THEN DEST[31:0] ← FFFFFFFFH;
            ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] > SRC[63:32]
            THEN DEST[63:32] ← FFFFFFFFH;
            ELSE DEST[63:32] ← 0; FI;
```

PCMPGTD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 encoded version)

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

VPCMPGTD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

VPCMPGTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
            THEN
                /* signed comparison */
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                        THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
                        ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
                FI;
                IF CMP = TRUE
                        THEN DEST[j] ← 1;
                        ELSE DEST[j] ← 0; FI;
            ELSE        DEST[j] ← 0                         ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
```

VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB:__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW:__m64 _mm_pcmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD:__m64 _mm_pcmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB:__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW:__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD:__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB: __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW: __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD: __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTD, see Exceptions Type E4.

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB *reg/m8, xmm2, imm8*	MRI	V/V	SSE4_1	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg or m8.* The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD *r/m32, xmm2, imm8*	MRI	V/V	SSE4_1	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r/m32.*
66 REX.W 0F 3A 16 /r ib PEXTRQ *r/m64, xmm2, imm8*	MRI	V/N.E.	SSE4_1	Extract a qword integer value from *xmm2* at the source qword offset specified by *imm8* into *r/m64.*
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB *reg/m8, xmm2, imm8*	MRI	V[1]/V	AVX	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg* or *m8*. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD *r32/m32, xmm2, imm8*	MRI	V/V	AVX	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r32/m32*.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ *r64/m64, xmm2, imm8*	MRI	V/I[2]	AVX	Extract a qword integer value from *xmm2* at the source dword offset specified by *imm8* into *r64/m64*.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	T1S-MRI	V/V	AVX512BW	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	T1S-MRI	V/V	AVX512DQ	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	T1S-MRI	V/N.E.[2]	AVX512DQ	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L"L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit

mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL ← COUNT[3:0];
            TEMP ← (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; };
            ELSE
                    R32[7:0] ← TEMP[7:0];
                    r32[31:8] ← ZERO_FILL; };
            FI;
    PEXTRD:SEL ← COUNT[1:0];
            TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
            DEST ← TEMP;
    PEXTRQ: SEL ← COUNT[0];
            TEMP ← (Src >> SEL*64);
            DEST ← TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset ← Imm8[0]
    r64/m64 ←(Src >> Src_Offset * 64)
ELSE
    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] ←   ZERO_FILL;
ELSE
    SRC_Offset ←. Imm8[3:0];
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] ←   ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB: int _mm_extract_epi8 (__m128i src, const int ndx);

PEXTRD: int _mm_extract_epi32 (__m128i src, const int ndx);

PEXTRQ: __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /*r* ib[1] PEXTRW *reg, mm, imm8*	RMI	V/V	SSE	Extract the word specified by *imm8* from *mm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /*r* ib PEXTRW *reg, xmm, imm8*	RMI	V/V	SSE2	Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW *reg/m16, xmm, imm8*	MRI	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW *reg, xmm1, imm8*	RMI	V[2]/V	AVX	Extract the word specified by *imm8* from *xmm1* and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW *reg/m16, xmm2, imm8*	MRI	V/V	AVX	Extract a word integer value from *xmm2* at the source word offset specified by *imm8* into *reg* or *m16*. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	RMI	V/V	AVX512BW	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	T1S-MRI	V/V	AVX512BW	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

Instruction Operand Encoding

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

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
          { SEL ← COUNT[1:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r64[15:0] ← TEMP[15:0];
            r64[63:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          { SEL ← COUNT[2:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r64[15:0] ← TEMP[15:0];
            r64[63:16] ← ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
          { SEL ← COUNT[1:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r32[15:0] ← TEMP[15:0];
            r32[31:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          { SEL ← COUNT[2:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r32[15:0] ← TEMP[15:0];
            r32[31:16] ← ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
```

VPEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] ←   ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] ←   ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW: int _mm_extract_pi16 (__m64 a, int n)

PEXTRW: int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 01 /r[1] PHADDW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *mm1*.
66 0F 38 01 /r PHADDW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *xmm1*.
NP 0F 38 02 /r PHADDD *mm1, mm2/m64*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *mm1*.
66 0F 38 02 /r PHADDD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to *ymm1*.
VEX.NDS.256.66.0F38.WIG 02 /r VPHADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

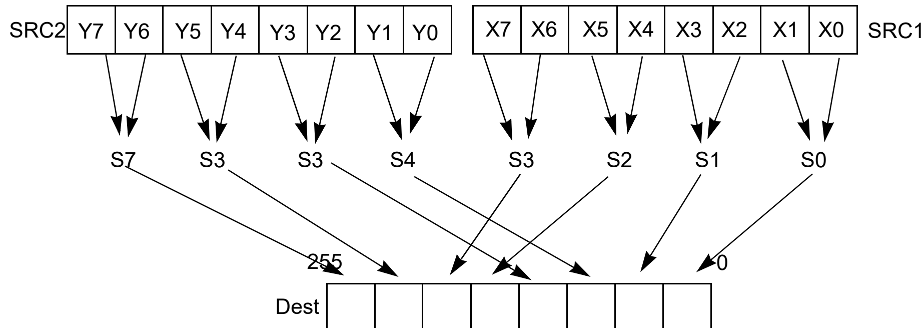


Figure 4-10. 256-bit VPHADDD Instruction Operation

Operation

PHADDW (with 64-bit operands)

```
mm1[15-0]  = mm1[31-16] + mm1[15-0];
mm1[31-16] = mm1[63-48] + mm1[47-32];
mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];
```

PHADDW (with 128-bit operands)

```
xmm1[15-0] = xmm1[31-16] + xmm1[15-0];
xmm1[31-16] = xmm1[63-48] + xmm1[47-32];
xmm1[47-32] = xmm1[95-80] + xmm1[79-64];
xmm1[63-48] = xmm1[127-112] + xmm1[111-96];
xmm1[79-64] = xmm2/m128[31-16] + xmm2/m128[15-0];
xmm1[95-80] = xmm2/m128[63-48] + xmm2/m128[47-32];
xmm1[111-96] = xmm2/m128[95-80] + xmm2/m128[79-64];
xmm1[127-112] = xmm2/m128[127-112] + xmm2/m128[111-96];
```

VPHADDW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[VLMAX-1:128] ← 0
```

VPHADDW (VEX.256 encoded version)
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[143:128] ← SRC1[159:144] + SRC1[143:128]
DEST[159:144] ← SRC1[191:176] + SRC1[175:160]
DEST[175:160] ← SRC1[223:208] + SRC1[207:192]
DEST[191:176] ← SRC1[255:240] + SRC1[239:224]
DEST[207:192] ← SRC2[127:112] + SRC2[143:128]
DEST[223:208] ← SRC2[159:144] + SRC2[175:160]
DEST[239:224] ← SRC2[191:176] + SRC2[207:192]
DEST[255:240] ← SRC2[223:208] + SRC2[239:224]

PHADDD (with 64-bit operands)
mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADDD (with 128-bit operands)
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

VPHADDD (VEX.128 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[VLMAX-1:128] ← 0

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

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW: __m64 _mm_hadd_pi16 (__m64 a, __m64 b)

PHADDD: __m64 _mm_hadd_pi32 (__m64 a, __m64 b)

(V)PHADDW: __m128i _mm_hadd_epi16 (__m128i a, __m128i b)

(V)PHADDD: __m128i _mm_hadd_epi32 (__m128i a, __m128i b)

VPHADDW: __m256i _mm256_hadd_epi16 (__m256i a, __m256i b)

VPHADDD: __m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHADDSW — Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 03 /r[1] PHADDSW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *mm1*.
66 0F 38 03 /r PHADDSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDSW (with 64-bit operands)

```
mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (with 128-bit operands)

```
xmm1[15-0]= SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
xmm1[63-48] = SaturateToSignedWord(xmm1[127-112] + xmm1[111-96]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
xmm1[127-112] = SaturateToSignedWord(xmm2/m128[127-112] + xmm2/m128[111-96]);
```

VPHADDSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[VLMAX-1:128] ← 0
```

VPHADDSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128]= SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord( SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW: __m64 _mm_hadds_pi16 (__m64 a, __m64 b)

(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

VPHADDSW: __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 05 /r[1] PHSUBW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *mm1*.
66 0F 38 05 /r PHSUBW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
NP 0F 38 06 /r PHSUBD *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *mm1*.
66 0F 38 06 /r PHSUBD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to *ymm1*.
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBW (with 64-bit operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

PHSUBW (with 128-bit operands)

```
xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];
```

VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[VLMAX-1:128] ← 0
```

VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

PHSUBD (with 64-bit operands)

```
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];
```

PHSUBD (with 128-bit operands)

```
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];
```

VPHSUBD (VEX.128 encoded version)

```
DEST[31-0] ← SRC1[31-0] - SRC1[63-32]
DEST[63-32] ← SRC1[95-64] - SRC1[127-96]
DEST[95-64] ← SRC2[31-0] - SRC2[63-32]
DEST[127-96] ← SRC2[95-64] - SRC2[127-96]
DEST[VLMAX-1:128] ← 0
```

VPHSUBD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW:	__m64 _mm_hsub_pi16 (__m64 a, __m64 b)
PHSUBD:	__m64 _mm_hsub_pi32 (__m64 a, __m64 b)
(V)PHSUBW:	__m128i _mm_hsub_epi16 (__m128i a, __m128i b)
(V)PHSUBD:	__m128i _mm_hsub_epi32 (__m128i a, __m128i b)
VPHSUBW:	__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
VPHSUBD:	__m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 07 /r[1] PHSUBSW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *mm1.*
66 0F 38 07 /r PHSUBSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1.*
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1*.
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to *ymm1*.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (with 128-bit operands)

```
xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
xmm1[31-16] = SaturateToSignedWord(xmm1[47-32] - xmm1[63-48]);
xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
xmm1[63-48] = SaturateToSignedWord(xmm1[111-96] - xmm1[127-112]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
xmm1[95-80] =SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
xmm1[111-96] =SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);
xmm1[127-112]= SaturateToSignedWord(xmm2/m128[111-96] - xmm2/m128[127-112]);
```

VPHSUBSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[VLMAX-1:128] ← 0
```

VPHSUBSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128]= SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW:	__m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW:	__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW:	__m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB *xmm1, r32/m8, imm8*	RMI	V/V	SSE4_1	Insert a byte integer value from *r32/m8* into *xmm1* at the destination element in *xmm1* specified by *imm8.*
66 0F 3A 22 /r ib PINSRD *xmm1, r/m32, imm8*	RMI	V/V	SSE4_1	Insert a dword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.*
66 REX.W 0F 3A 22 /r ib PINSRQ *xmm1, r/m64, imm8*	RMI	V/N. E.	SSE4_1	Insert a qword integer value from *r/m64 i*nto the *xmm1* at the destination element specified by *imm8.*
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB *xmm1, xmm2, r32/m8, imm8*	RVMI	V[1]/V	AVX	Merge a byte integer value from *r32/m8* and rest from *xmm2* into *xmm1* at the byte offset in *imm8*.
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD *xmm1, xmm2, r/m32, imm8*	RVMI	V/V	AVX	Insert a dword integer value from *r32/m32* and rest from *xmm2* into *xmm1* at the dword offset in *imm8*.
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ *xmm1, xmm2, r/m64, imm8*	RVMI	V/I[2]	AVX	Insert a qword integer value from *r64/m64* and rest from *xmm2* into *xmm1* at the qword offset in *imm8*.
EVEX.NDS.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	T1S-RVMI	V/V	AVX512BW	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	T1S-RVMI	V/V	AVX512DQ	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	T1S-RVMI	V/N.E.[2]	AVX512DQ	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
T1S-RVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL ← COUNT[3:0];
             MASK ← (0FFH << (SEL * 8));
             TEMP ← (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL ← COUNT[1:0];
             MASK ← (0FFFFFFFFH << (SEL * 32));
             TEMP ← (((SRC << (SEL *32)) AND MASK)  ;
    PINSRQ:  SEL ← COUNT[0]
             MASK ← (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP ← (((SRC << (SEL *64)) AND MASK)  ;
ESAC;
        DEST ← ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRD (VEX/EVEX encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRQ (VEX/EVEX encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PINSRB:    __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:    __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:    __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L’L > 0.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C4 /*r* ib[1] PINSRW *mm, r32/m16, imm8*	RMI	V/V	SSE	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8.*
66 0F C4 /*r* ib PINSRW *xmm, r32/m16, imm8*	RMI	V/V	SSE2	Move the low word of *r32* or from *m16* into *xmm* at the word position specified by *imm8.*
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW *xmm1, xmm2, r32/m16, imm8*	RVMI	V[2]/V	AVX	Insert a word integer value from *r32/m16* and rest from *xmm2* into *xmm1* at the word offset in *imm8.*
EVEX.NDS.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	T1S-RVMI	V/V	AVX512BW	Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.*
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
T1S-RVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
    CASE (Determine word position) OF
        SEL ← 0:      MASK ← 000000000000FFFFH;
```

```
            SEL ← 1:      MASK ← 00000000FFFF0000H;
            SEL ← 2:      MASK ← 0000FFFF00000000H;
            SEL ← 3:      MASK ← FFFF000000000000H;
    DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);
```

PINSRW (with 128-bit source operand)

```
    SEL ← COUNT AND 7H;
        CASE (Determine word position) OF
            SEL ← 0:      MASK ← 0000000000000000000000000000FFFFH;
            SEL ← 1:      MASK ← 000000000000000000000000FFFF0000H;
            SEL ← 2:      MASK ← 00000000000000000000FFFF00000000H;
            SEL ← 3:      MASK ← 0000000000000000FFFF000000000000H;
            SEL ← 4:      MASK ← 000000000000FFFF0000000000000000H;
            SEL ← 5:      MASK ← 00000000FFFF00000000000000000000H;
            SEL ← 6:      MASK ← 0000FFFF000000000000000000000000H;
            SEL ← 7:      MASK ← FFFF0000000000000000000000000000H;
    DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);
```

VPINSRW (VEX/EVEX encoded version)

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW: __m64 _mm_insert_pi16 (__m64 a, int d, int n)

PINSRW: __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L'L > 0.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r[1] PMADDUBSW *mm1, mm2/m64*	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *mm1*.
66 0F 38 04 /r PMADDUBSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *ymm1*.
EVEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (with 64 bit operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
    DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
    DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
    DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (with 128 bit operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
    // Repeat operation for 2nd through 7th word
    SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[VLMAX-1:128] ← 0
```

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[VLMAX-1:256] ← 0
```

VPMADDUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_mddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_mddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_mddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_mddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_mddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_mddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW: __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW: __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD *mm, mm/m64*	RM	V/V	MMX	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r PMADDWD *xmm1, xmm2/m128*	RM	V/V	SSE2	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply the packed word integers in *xmm2* by the packed word integers in *xmm3/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply the packed word integers in *ymm2* by the packed word integers in *ymm3/m256*, add adjacent doubleword results, and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

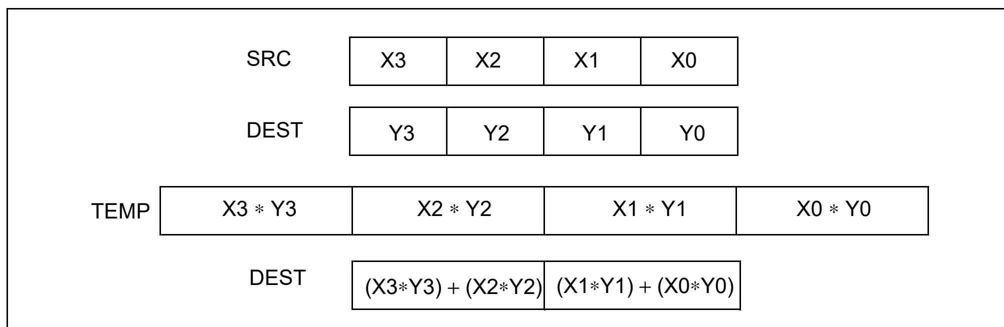


Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

```
DEST[31:0] ← (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
DEST[63:32] ← (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
```

PMADDWD (with 128-bit operands)

```
DEST[31:0] ← (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
DEST[63:32] ← (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
DEST[95:64] ← (DEST[79:64] ∗ SRC[79:64]) + (DEST[95:80] ∗ SRC[95:80]);
DEST[127:96] ← (DEST[111:96] ∗ SRC[111:96]) + (DEST[127:112] ∗ SRC[127:112]);
```

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[VLMAX-1:128] ← 0
```

VPMADDWD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
```

```
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[VLMAX-1:256] ← 0
```

VPMADDWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADDWD __m512i _mm512_mdd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_mdd_epi16(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_mdd_epi16( __mmask16 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_mdd_epi16(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_mdd_epi16( __mmask8 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_mdd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDWD:__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD:__m128i _mm_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD:__m256i _mm256_madd_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /r[1] PMAXSW *mm1, mm2/m64*	RM	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit operands)

```
IF DEST[15:0] > SRC[15:0]) THEN
    DEST[15:0] ← DEST[15:0];
```

```
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] > SRC[63:48]) THEN
        DEST[63:48] ← DEST[63:48];
    ELSE
        DEST[63:48] ← SRC[63:48]; FI;
```

PMAXSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ←DEST[7:0];
    ELSE
        DEST[7:0] ←SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ←DEST[127:120];
    ELSE
        DEST[127:120] ←SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ←SRC1[7:0];
    ELSE
        DEST[7:0] ←SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ←SRC1[127:120];
    ELSE
        DEST[127:120] ←SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ←0
```

VPMAXSB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ←SRC1[7:0];
    ELSE
        DEST[7:0] ←SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ←SRC1[255:248];
    ELSE
        DEST[255:248] ←SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ←0
```

VPMAXSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
```

```
                IF *merging-masking*                    ; merging-masking
                      THEN *DEST[i+7:i] remains unchanged*
                      ELSE                              ; zeroing-masking
                            DEST[i+7:i] ← 0
                FI
     FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PMAXSW (128-bit Legacy SSE version)

```
     IF DEST[15:0] >SRC[15:0] THEN
           DEST[15:0] ←DEST[15:0];
     ELSE
           DEST[15:0] ←SRC[15:0]; FI;
     (* Repeat operation for 2nd through 7th words in source and destination operands *)
     IF DEST[127:112] >SRC[127:112] THEN
           DEST[127:112] ←DEST[127:112];
     ELSE
           DEST[127:112] ←SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 encoded version)

```
     IF SRC1[15:0] > SRC2[15:0] THEN
           DEST[15:0] ←SRC1[15:0];
     ELSE
           DEST[15:0] ←SRC2[15:0]; FI;
     (* Repeat operation for 2nd through 7th words in source and destination operands *)
     IF SRC1[127:112] >SRC2[127:112] THEN
           DEST[127:112] ←SRC1[127:112];
     ELSE
           DEST[127:112] ←SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ←0
```

VPMAXSW (VEX.256 encoded version)

```
     IF SRC1[15:0] > SRC2[15:0] THEN
           DEST[15:0] ←SRC1[15:0];
     ELSE
           DEST[15:0] ←SRC2[15:0]; FI;
     (* Repeat operation for 2nd through 15th words in source and destination operands *)
     IF SRC1[255:240] >SRC2[255:240] THEN
           DEST[255:240] ←SRC1[255:240];
     ELSE
           DEST[255:240] ←SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ←0
```

VPMAXSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
     i ← j * 16
     IF k1[j] OR *no writemask* THEN
           IF SRC1[i+15:i] > SRC2[i+15:i]
                 THEN DEST[i+15:i] ← SRC1[i+15:i];
                 ELSE DEST[i+15:i] ← SRC2[i+15:i];
           FI;
           ELSE
```

```
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PMAXSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ←DEST[31:0];
    ELSE
        DEST[31:0] ←SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] ←DEST[127:96];
    ELSE
        DEST[127:96] ←SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] ←SRC1[127:96];
    ELSE
        DEST[127:96] ←SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ←0
```

VPMAXSD (VEX.256 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] ←SRC1[255:224];
    ELSE
        DEST[255:224] ←SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ←0
```

VPMAXSD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
```

```
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPMAXSQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMAXSB __m512i _mm512_max_epi8( __m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8( __mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16( __m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16( __mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8( __mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
```

VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPMAXSD/Q, see Exceptions Type E4.

EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb.

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB *mm1, mm2/m64*	RM	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.NDS.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.NDS.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit operands)

```
    IF DEST[7:0] > SRC[17:0]) THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[7:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] > SRC[63:56]) THEN
        DEST[63:56] ← DEST[63:56];
    ELSE
        DEST[63:56] ← SRC[63:56]; FI;
```

PMAXUB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMAXUB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMAXUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PMAXUW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMAXUW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMAXUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMAXUB __m512i _mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8( __mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16( __mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8( __mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16( __mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8( __mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 ( __m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 ( __m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 ( __m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 ( __m256i a, __m256i b);
PMAXUB: __m64 _mm_max_pu8(__m64 a, __m64 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW *mm1, mm2/m64*	RM	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] < SRC[63:48] THEN
        DEST[63:48] ← DEST[63:48];
    ELSE
        DEST[63:48] ← SRC[63:48]; FI;
```

PMINSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMINSB (VEX.128 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMINSB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

VPMINSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PMINSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMINSW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMINSW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ← 0
```

VPMINSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINSB __m512i _mm512_min_epi8( __m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8( __mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16( __m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16( __mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8( __mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16( __mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8( __mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16( __mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 ( __m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 ( __m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 ( __m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 ( __m256i a, __m256i b)
PMINSW:__m64 _mm_min_pi16 (__m64 a, __m64 b)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB *mm1, mm2/m64*	RM	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUB (for 64-bit operands)

```
    IF DEST[7:0] < SRC[17:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[7:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] < SRC[63:56] THEN
        DEST[63:56] ← DEST[63:56];
    ELSE
        DEST[63:56] ← SRC[63:56]; FI;
```

PMINUB instruction for 128-bit operands:

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMINUB (VEX.128 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMINUB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

VPMINUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

PMINUW instruction for 128-bit operands:

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

VPMINUW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0
```

VPMINUW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
          DEST[15:0] ← SRC1[15:0];
    ELSE
          DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
          DEST[255:240] ← SRC1[255:240];
    ELSE
          DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ← 0
```

VPMINUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
          IF SRC1[i+15:i] < SRC2[i+15:i]
                THEN DEST[i+15:i] ← SRC1[i+15:i];
                ELSE DEST[i+15:i] ← SRC2[i+15:i];
          FI;
          ELSE
                IF *merging-masking*                        ; merging-masking
                      THEN *DEST[i+15:i] remains unchanged*
                      ELSE                                  ; zeroing-masking
                            DEST[i+15:i] ← 0
                FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINUB __m512i _mm512_min_epu8( __m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8( __mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16( __m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16( __mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8( __mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16( __mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8( __mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 ( __m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 ( __m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 ( __m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 ( __m256i a, __m256i b);
PMINUB: __m64 _m_min_pu8 (__m64 a, __m64 b)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D7 /r[1] PMOVMSKB *reg, mm*	RM	V/V	SSE	Move a byte mask of *mm* to *reg*. The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMSKB *reg, xmm*	RM	V/V	SSE2	Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMSKB *reg, xmm1*	RM	V/V	AVX	Move a byte mask of *xmm1* to *reg*. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMSKB *reg, ymm1*	RM	V/V	AVX2	Move a 32-bit mask of *ymm1* to *reg*. The upper bits of r64 are filled with zeros.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

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

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (with 64-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

(V)PMOVMSKB (with 128-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;
```

VPMOVMSKB (with 256-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 3rd through 31*)
r32[31] ← SRC[255];
```

PMOVMSKB (with 64-bit source operand and r64)

```
    r64[0] ← SRC[7];
    r64[1] ← SRC[15];
    (* Repeat operation for bytes 2 through 6 *)
    r64[7] ← SRC[63];
    r64[63:8] ← ZERO_FILL;
```

(V)PMOVMSKB (with 128-bit source operand and r64)

```
    r64[0] ← SRC[7];
    r64[1] ← SRC[15];
    (* Repeat operation for bytes 2 through 14 *)
    r64[15] ← SRC[127];
    r64[63:16] ← ZERO_FILL;
```

VPMOVMSKB (with 256-bit source operand and r64)

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB:	int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB:	int _mm_movemask_epi8 (__m128i a)
VPMOVMSKB:	int _mm256_movemask_epi8 (__m256i a)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW *mm1, mm2/m64*	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *mm1*.
66 0F 38 0B /r PMULHRSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *ymm1*.
EVEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHRSW (with 64-bit operands)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
```

PMULHRSW (with 128-bit operand)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
    temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
    temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
    temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
    temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
    DEST[79:64] = temp4[16:1];
    DEST[95:80] = temp5[16:1];
    DEST[111:96] = temp6[16:1];
    DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

```
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX-1:128] ← 0
```

VPMULHRSW (VEX.256 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
DEST[MAX_VL-1:256] ← 0
```

VPMULHRSW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] ← tmp[16:1]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
```

```
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW: __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW:__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW *mm1, mm2/m64*	RM	V/V	SSE	Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E4 /r PMULHUW *xmm1, xmm2/m128*	RM	V/V	SSE2	Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply the packed unsigned word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply the packed unsigned word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

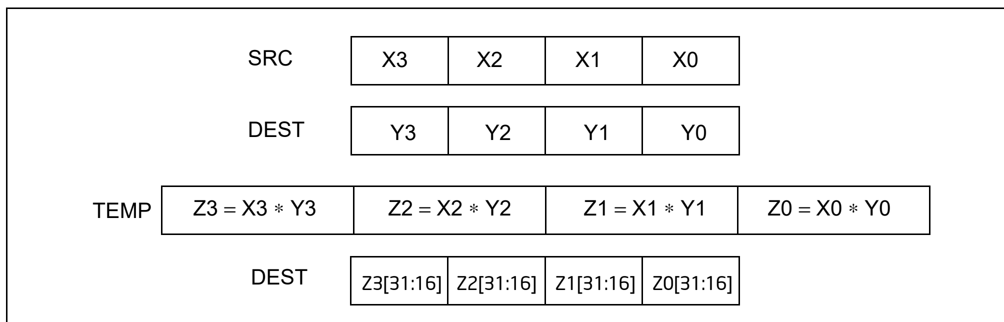


Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (with 64-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] ←     TEMP0[31:16];
    DEST[31:16] ←    TEMP1[31:16];
    DEST[47:32] ←    TEMP2[31:16];
    DEST[63:48] ←    TEMP3[31:16];
```

PMULHUW (with 128-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] ←    DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] ←    DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] ←    DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] ←    DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] ←     TEMP0[31:16];
    DEST[31:16] ←    TEMP1[31:16];
    DEST[47:32] ←    TEMP2[31:16];
    DEST[63:48] ←    TEMP3[31:16];
    DEST[79:64] ←    TEMP4[31:16];
    DEST[95:80] ←    TEMP5[31:16];
    DEST[111:96] ←   TEMP6[31:16];
    DEST[127:112] ← TEMP7[31:16];
```

VPMULHUW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
```

```
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0
```

PMULHUW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAX_VL-1:256] ← 0
```

PMULHUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
```

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
         THEN
              temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
              DEST[i+15:i] ← tmp[31:16]
         ELSE
              IF *merging-masking*                ; merging-masking
                   THEN *DEST[i+15:i] remains unchanged*
                   ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] ← 0
              FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16( __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16( __mmask8 k, __m128i a, __m128i b);
PMULHUW:__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW:__m128i _mm_mulhi_epu16 ( __m128i a, __m128i b)
VPMULHUW:__m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW *mm, mm/m64*	RM	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /r PMULHW *xmm1, xmm2/m128*	RM	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply the packed signed word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (with 64-bit operands)

```
TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
DEST[15:0] ←     TEMP0[31:16];
DEST[31:16] ←    TEMP1[31:16];
DEST[47:32] ←    TEMP2[31:16];
DEST[63:48] ←    TEMP3[31:16];
```

PMULHW (with 128-bit operands)

```
TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
TEMP4[31:0] ←    DEST[79:64] ∗ SRC[79:64];
TEMP5[31:0] ←    DEST[95:80] ∗ SRC[95:80];
TEMP6[31:0] ←    DEST[111:96] ∗ SRC[111:96];
TEMP7[31:0] ←    DEST[127:112] ∗ SRC[127:112];
DEST[15:0] ←     TEMP0[31:16];
DEST[31:16] ←    TEMP1[31:16];
DEST[47:32] ←    TEMP2[31:16];
DEST[63:48] ←    TEMP3[31:16];
DEST[79:64] ←    TEMP4[31:16];
DEST[95:80] ←    TEMP5[31:16];
DEST[111:96] ←   TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];
```

VPMULHW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0
```

PMULHW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[VLMAX-1:256] ← 0
```

PMULHW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW:__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW:__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW:__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /r[1] PMULLW *mm, mm/m64*	RM	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /r PMULLW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply the packed dword signed integers in *xmm2* and *xmm3/m128* and store the low 32 bits of each product in *xmm1*.
VEX.NDS.256.66.0F.WIG D5 /r VPMULLW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the low 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

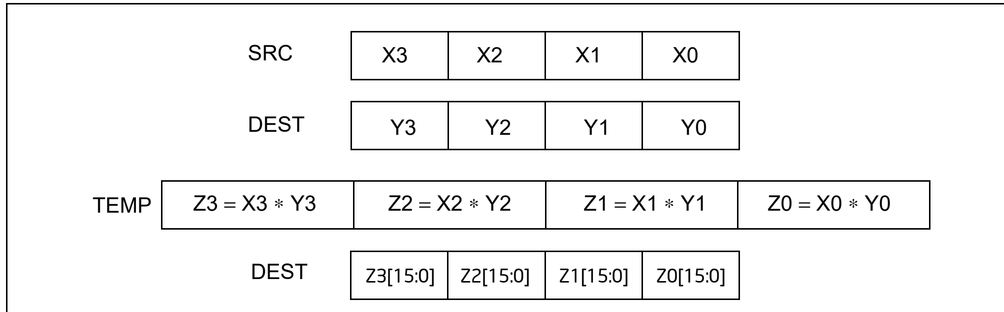


Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (with 64-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] ←     TEMP0[15:0];
    DEST[31:16] ←    TEMP1[15:0];
    DEST[47:32] ←    TEMP2[15:0];
    DEST[63:48] ←    TEMP3[15:0];
```

PMULLW (with 128-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] ←    DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] ←    DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] ←    DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] ←    DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] ←     TEMP0[15:0];
    DEST[31:16] ←    TEMP1[15:0];
    DEST[47:32] ←    TEMP2[15:0];
    DEST[63:48] ←    TEMP3[15:0];
    DEST[79:64] ←    TEMP4[15:0];
    DEST[95:80] ←    TEMP5[15:0];
    DEST[111:96] ←   TEMP6[15:0];
    DEST[127:112] ← TEMP7[15:0];
DEST[VLMAX-1:256] ← 0
```

VPMULLW (VEX.128 encoded version)

```
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[VLMAX-1:128] ← 0
```

PMULLW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← temp[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
PMULLW: __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW: __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW:__m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F4 /r[1] PMULUDQ *mm1*, *mm2/m64*	RM	V/V	SSE2	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r PMULUDQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply packed unsigned doubleword integers in *xmm2* by packed unsigned doubleword integers in *xmm3/m128*, and store the quadword results in *xmm1*.
VEX.NDS.256.66.0F.WIG F4 /r VPMULUDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply packed unsigned doubleword integers in *ymm2* by packed unsigned doubleword integers in *ymm3/m256*, and store the quadword results in *ymm1*.
EVEX.NDS.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned

quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (with 64-Bit operands)

```
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];
```

PMULUDQ (with 128-Bit operands)

```
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];
    DEST[127:64] ← DEST[95:64] ∗ SRC[95:64];
```

VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[VLMAX-1:128] ← 0
```

VPMULUDQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]
DEST[VLMAX-1:256] ← 0
```

VPMULUDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
```

```
                THEN DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
                ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i] )
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32( __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32( __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32( __mmask8 k, __m128i a, __m128i b);
PMULUDQ:__m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ:__m128i _mm_mul_epu32 ( __m128i a, __m128i b)
VPMULUDQ:__m256i _mm256_mul_epu32( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /*r*[1] POR *mm, mm/m64*	RM	V/V	MMX	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /*r* POR *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EB /r VPOR *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise OR of *xmm2/m128* and *xmm3*.
VEX.NDS.256.66.0F.WIG EB /r VPOR *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise OR of *ymm2/m256* and *ymm3*.
EVEX.NDS.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.NDS.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.NDS.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.NDS.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.NDS.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

Operation

POR (64-bit operand)

```
DEST ← DEST OR SRC
```

POR (128-bit Legacy SSE version)

```
DEST ← DEST OR SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPOR (VEX.128 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[VLMAX-1:128] ← 0
```

VPOR (VEX.256 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[VLMAX-1:256] ← 0
```

VPORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
               ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
      ELSE
            IF *merging-masking*                    ; merging-masking
               *DEST[i+31:i] remains unchanged*
               ELSE                                 ; zeroing-masking
                  DEST[i+31:i] ← 0
            FI;
   FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32(__mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64(__mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64(__mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR: __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR: __m256i _mm256_or_si256 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW *mm1, mm2/m64*	RM	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW *xmm1, xmm2/m128*	RM	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from *xmm3 /m128* and *xmm2*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from *ymm3 /m256* and *ymm2*; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second

operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

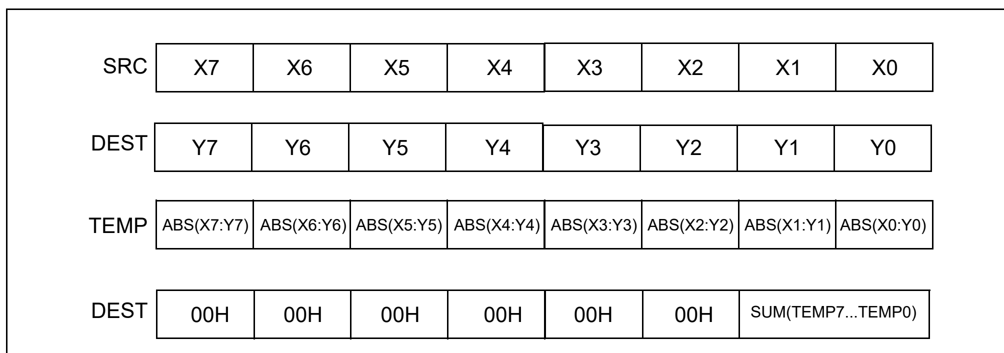


Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX encoded versions)

```
VL = 128, 256, 512
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
```

```
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] ←SUM(TEMP16:TEMP23)
    DEST[191:144] ← 000000000000H
    DEST[207:192] ← SUM(TEMP24:TEMP31)
    DEST[223:208] ← 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] ←SUM(TEMP0:TEMP7)
    DEST[319:272] ← 000000000000H
    DEST[335:320] ← SUM(TEMP8:TEMP15)
    DEST[383:336] ← 00000000000H
    DEST[399:384] ←SUM(TEMP16:TEMP23)
    DEST[447:400] ← 000000000000H
    DEST[463:448] ← SUM(TEMP24:TEMP31)
    DEST[511:464] ← 00000000000H
FI;
DEST[MAX_VL-1:VL] ← 0
```

VPSADBW (VEX.256 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[143:128] ←SUM(TEMP16:TEMP23)
DEST[191:144] ← 000000000000H
DEST[207:192] ← SUM(TEMP24:TEMP31)
DEST[223:208] ← 00000000000H
DEST[MAX_VL-1:256] ← 0
```

VPSADBW (VEX.128 encoded version)
```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[MAX_VL-1:128] ← 0
```

PSADBW (128-bit Legacy SSE version)
```
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000
DEST[MAX_VL-1:128] (Unmodified)
```

PSADBW (64-bit operand)
```
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSADBW __m512i _mm512_sad_epu8( __m512i a, __m512i b)
PSADBW:__m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW:__m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW:__m256i _mm256_sad_epu8( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 00 /r[1] PSHUFB *mm1, mm2/m64*	RM	V/V	SSSE3	Shuffle bytes in *mm1* according to contents of *mm2/m64*.
66 0F 38 00 /r PSHUFB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Shuffle bytes in *xmm1* according to contents of *xmm2/m128*.
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shuffle bytes in *xmm2* according to contents of *xmm3/m128*.
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shuffle bytes in *ymm2* according to contents of *ymm3/m256*.
EVEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.
EVEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.
EVEX.NDS.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

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

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.

The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

Operation

PSHUFB (with 64 bit operands)

```
TEMP ← DEST
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif;
}
```

PSHUFB (with 128 bit operands)

```
TEMP ← DEST
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
     else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0
```

VPSHUFB (VEX.256 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
```

```
    endif
}
VPSHUFB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask ← (KL-1) & ~0xF                          // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                               // dest
    IF kl[ i ] or no_masking
        index ← src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] ← 0;
        ELSE
            index ← (index & 0xF) + (j & jmask);    // 16-element in-lane lookup
            Dest.byte[ j ] ← src.byte[ index ];
    ELSE if zeroing
        Dest.byte[ j ] ← 0;
DEST[MAX_VL-1:VL] ← 0;
```

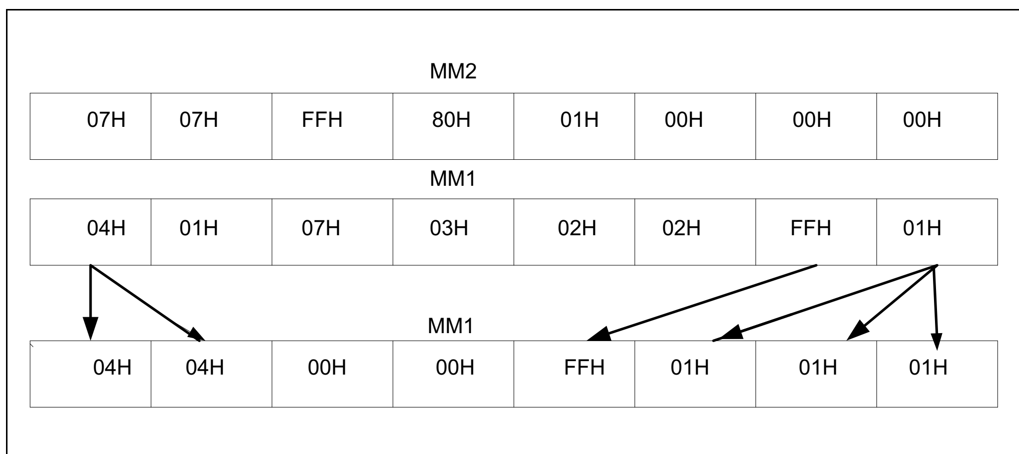


Figure 4-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFW—Shuffle Packed Words

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

Instruction Operand Encoding

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

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

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

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 22-7, “Exception Conditions for SIMD/MMX Instructions with Memory Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 08 /r[1] PSIGNB *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *mm1* depending on the corresponding sign in *mm2/m64.*
66 0F 38 08 /r PSIGNB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *xmm1* depending on the corresponding sign in *xmm2/m128.*
NP 0F 38 09 /r[1] PSIGNW *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *mm1* depending on the corresponding sign in *mm2/m128.*
66 0F 38 09 /r PSIGNW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *xmm1* depending on the corresponding sign in *xmm2/m128.*
NP 0F 38 0A /r[1] PSIGND *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *mm1* depending on the corresponding sign in *mm2/m128.*
66 0F 38 0A /r PSIGND *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *xmm1* depending on the corresponding sign in *xmm2/m128.*
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in *xmm2* depending on the corresponding sign in *xmm3/m128.*
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed word integers in *xmm2* depending on the corresponding sign in *xmm3/m128.*
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in *xmm2* depending on the corresponding sign in *xmm3/m128.*
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed byte integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed 16-bit integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed doubleword integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
      DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
      DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
      DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

IF (SRC[63:56] < 0 )
      DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
      DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
      DEST[63:56] ← DEST[63:56]
```

PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
      DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
      DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
      DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
      DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
      DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
      DEST[127:120] ← DEST[127:120]
```

VPSIGNB (VEX.128 encoded version)

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

VPSIGNB (VEX.256 encoded version)

```
DEST[255:0] ←BYTE_SIGN_256b(SRC1, SRC2)
```

PSIGNW (with 64 bit operands)

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
    IF (SRC[63:48] < 0 )
        DEST[63:48] ← Neg(DEST[63:48])
    ELSEIF (SRC[63:48] = 0 )
        DEST[63:48] ← 0
    ELSEIF (SRC[63:48] > 0 )
        DEST[63:48] ← DEST[63:48]
```

PSIGNW (with 128 bit operands)

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
    Repeat operation for 2nd through 7th words
    IF (SRC[127:112] < 0 )
        DEST[127:112] ← Neg(DEST[127:112])
    ELSEIF (SRC[127:112] = 0 )
        DEST[127:112] ← 0
    ELSEIF (SRC[127:112] > 0 )
        DEST[127:112] ← DEST[127:112]
```

VPSIGNW (VEX.128 encoded version)

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

VPSIGNW (VEX.256 encoded version)

```
DEST[255:0] ←WORD_SIGN(SRC1, SRC2)
```

PSIGND (with 64 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    IF (SRC[63:32] < 0 )
        DEST[63:32] ← Neg(DEST[63:32])
    ELSEIF (SRC[63:32] = 0 )
        DEST[63:32] ← 0
```

```
    ELSEIF (SRC[63:32] > 0 )
        DEST[63:32] ← DEST[63:32]
```

PSIGND (with 128 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    Repeat operation for 2nd through 3rd double words
    IF (SRC[127:96] < 0 )
        DEST[127:96] ← Neg(DEST[127:96])
    ELSEIF (SRC[127:96] = 0 )
        DEST[127:96] ← 0
    ELSEIF (SRC[127:96] > 0 )
        DEST[127:96] ← DEST[127:96]
```

VPSIGND (VEX.128 encoded version)

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

VPSIGND (VEX.256 encoded version)

```
DEST[255:0] ←DWORD_SIGN(SRC1, SRC2)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB: __m64 _mm_sign_pi8 (__m64 a, __m64 b)

(V)PSIGNB: __m128i _mm_sign_epi8 (__m128i a, __m128i b)

VPSIGNB: __m256i _mm256_sign_epi8 (__m256i a, __m256i b)

PSIGNW: __m64 _mm_sign_pi16 (__m64 a, __m64 b)

(V)PSIGNW: __m128i _mm_sign_epi16 (__m128i a, __m128i b)

VPSIGNW: __m256i _mm256_sign_epi16 (__m256i a, __m256i b)

PSIGND: __m64 _mm_sign_pi32 (__m64 a, __m64 b)

(V)PSIGND: __m128i _mm_sign_epi32 (__m128i a, __m128i b)

VPSIGND: __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW *mm, mm/m64*	RM	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r PSLLW *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 71 /6 ib PSLLW *mm1, imm8*	MI	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1, imm8*	MI	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F2 /r[1] PSLLD *mm, mm/m64*	RM	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r PSLLD *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1, imm8*	MI	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F3 /r[1] PSLLQ *mm, mm/m64*	RM	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r PSLLQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ *mm, imm8*	MI	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1, imm8*	MI	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift quadwords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift quadwords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* left by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift quadwords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift quadwords in *ymm2* left by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512BW	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	FVMI	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	FVMI	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	FVMI	V/V	AVX512BW	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.NDS.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVMI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

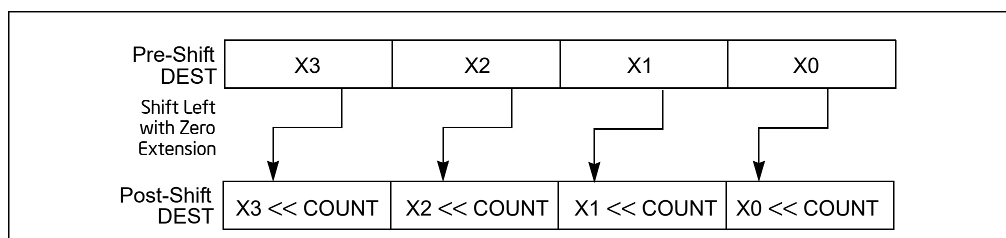


Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
```

PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST ← ZeroExtend(DEST << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
```

```
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
```

```
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSLLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSLLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSLLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSLLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSLLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSLLD (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSLLQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask* THEN
           IF (EVEX.b = 1) AND (SRC1 *is memory*)
               THEN DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
               ELSE DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
           FI;
       ELSE
           IF *merging-masking*              ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE *zeroing-masking*            ; zeroing-masking
                   DEST[i+63:i] ← 0
           FI
   FI;
ENDFOR
```

VPSLLQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
   TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
   TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
   TMP_DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
   TMP_DEST[511:256] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ←0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSLLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSLLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32( __mmask8 k, __m128i a, __m128i cnt);
```

VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_sllii_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW:__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW:__m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW:__m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW:__m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW:__m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW:__m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD:__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD:__m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD:__m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD:__m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD:__m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD:__m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ:__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ:__m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ:__m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ:__m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ:__m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ:__m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSLLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW *mm, mm/m64*	RM	V/V	MMX	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /r PSRAW *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW *mm, imm8*	MI	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib PSRAW *xmm1*, imm8	MI	V/V	SSE2	Shift words in *xmm1* right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD *mm, mm/m64*	RM	V/V	MMX	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /r PSRAD *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib PSRAD *xmm1*, imm8	MI	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in sign bits.
EVEX.NDS.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

EVEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	FVMI	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	FVMI	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	FVMI	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FVI	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVMI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

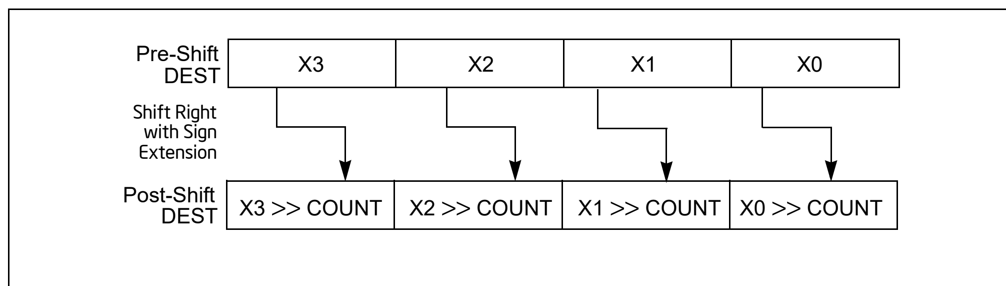


Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (with 64-bit operand)

```
    IF (COUNT > 15)
        THEN COUNT ← 16;
    FI;
    DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

PSRAD (with 64-bit operand)

```
    IF (COUNT > 31)
        THEN COUNT ← 32;
    FI;
    DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← SignBit
ELSE
    DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← SignBit
ELSE
    DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)            ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);


ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRAW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX
```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0
```

VPSRAW (ymm, imm8) - VEX
```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0
```

VPSRAW (xmm, imm8) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0
```

PSRAW (xmm, xmm, xmm/m128)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRAD (EVEX versions, imm8)
```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRAD (EVEX versions, xmm/m128)
```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
```

```
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRAQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRAQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] ← ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64( __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64( __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
```

VPSRAW __m512i _mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16(__mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16(__mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRAW:__m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW:__m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW:__m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW:__m128i _mm_sra_epi16(__m128i m, __m128i count)
VPSRAW:__m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW:__m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD:__m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD:__m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD:__m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD:__m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD:__m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD:__m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRAW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW *mm, mm/m64*	RM	V/V	MMX	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r PSRLW *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW *mm, imm8*	MI	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	MI	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D2 /r[1] PSRLD *mm, mm/m64*	RM	V/V	MMX	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r PSRLD *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	MI	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D3 /r[1] PSRLQ *mm, mm/m64*	RM	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /r PSRLQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ *mm, imm8*	MI	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	MI	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift quadwords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift quadwords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift quadwords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift quadwords in *ymm2* right by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	FVM	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVM	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

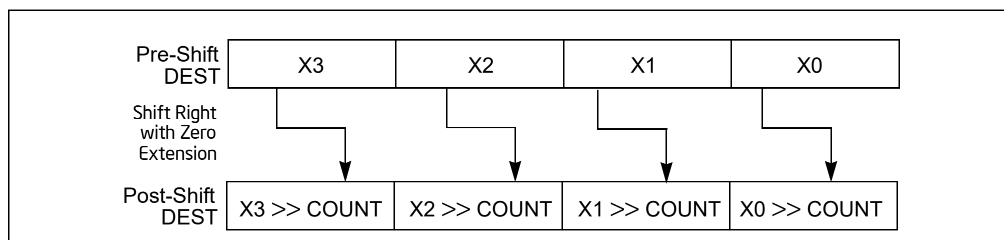


Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (with 64-bit operand)

```
    IF (COUNT > 15)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
    FI;
```

PSRLD (with 64-bit operand)

```
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (with 64-bit operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST ← ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
```

```
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ←0
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ←0
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ←0
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSRLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSRLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRLD (xmm, imm8) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRLD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

VPSRLQ (EVEX versions, xmm/m128)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLQ (EVEX versions, imm8)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC1 *is memory*)
                    THEN DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                    ELSE DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
                FI;
        ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding
```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0;
```

VPSRLQ (ymm, imm8) - VEX.256 encoding
```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ←0;
```
VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPSRLQ (xmm, imm8) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ←0
```

PSRLQ (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents
```
VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i _mm256_maskz_srli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
```

VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srlii_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW:__m64 _mm_srli_pi16(__m64 m, int count)
PSRLW:__m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW:__m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW:__m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW:__m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW:__m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD:__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD:__m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD:__m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD:__m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD:__m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD:__m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ:__m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ:__m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ:__m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ:__m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ:__m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ:__m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB *mm, mm/m64*	RM	V/V	MMX	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /r PSUBB *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
NP 0F F9 /r[1] PSUBW *mm, mm/m64*	RM	V/V	MMX	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /r PSUBW *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
NP 0F FA /r[1] PSUBD *mm, mm/m64*	RM	V/V	MMX	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /r PSUBD *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed byte integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed word integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed doubleword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed byte integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed word integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG FA /r VPSUBD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed doubleword integers in *ymm3/m256* from *ymm2*.
EVEX.NDS.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

EVEX.NDS.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (with 64-bit operands)

```
    DEST[7:0] ← DEST[7:0] − SRC[7:0];
    (* Repeat subtract operation for 2nd through 7th byte *)
    DEST[63:56] ← DEST[63:56] − SRC[63:56];
```

PSUBW (with 64-bit operands)

```
    DEST[15:0] ← DEST[15:0] − SRC[15:0];
    (* Repeat subtract operation for 2nd and 3rd word *)
    DEST[63:48] ← DEST[63:48] − SRC[63:48];
```

PSUBD (with 64-bit operands)

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

PSUBD (with 128-bit operands)

```
    DEST[31:0]  ← DEST[31:0] − SRC[31:0];
    (* Repeat subtract operation for 2nd and 3rd doubleword *)
    DEST[127:96] ← DEST[127:96] − SRC[127:96];
```

VPSUBB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] - SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPSUBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] - SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPSUBD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPSUBB (VEX.256 encoded version)

```
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[135:128] ←SRC1[135:128]-SRC2[135:128]
DEST[143:136] ←SRC1[143:136]-SRC2[143:136]
DEST[151:144] ←SRC1[151:144]-SRC2[151:144]
DEST[159:152] ←SRC1[159:152]-SRC2[159:152]
DEST[167:160] ←SRC1[167:160]-SRC2[167:160]
DEST[175:168] ←SRC1[175:168]-SRC2[175:168]
DEST[183:176] ←SRC1[183:176]-SRC2[183:176]
DEST[191:184] ←SRC1[191:184]-SRC2[191:184]
DEST[199:192] ←SRC1[199:192]-SRC2[199:192]
DEST[207:200] ←SRC1[207:200]-SRC2[207:200]
```

DEST[215:208] ←SRC1[215:208]-SRC2[215:208]
DEST[223:216] ←SRC1[223:216]-SRC2[223:216]
DEST[231:224] ←SRC1[231:224]-SRC2[231:224]
DEST[239:232] ←SRC1[239:232]-SRC2[239:232]
DEST[247:240] ←SRC1[247:240]-SRC2[247:240]
DEST[255:248] ←SRC1[255:248]-SRC2[255:248]
DEST[MAX_VL-1:256] ←0

VPSUBB (VEX.128 encoded version)
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[MAX_VL-1:128] ←0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ←DEST[7:0]-SRC[7:0]
DEST[15:8] ←DEST[15:8]-SRC[15:8]
DEST[23:16] ←DEST[23:16]-SRC[23:16]
DEST[31:24] ←DEST[31:24]-SRC[31:24]
DEST[39:32] ←DEST[39:32]-SRC[39:32]
DEST[47:40] ←DEST[47:40]-SRC[47:40]
DEST[55:48] ←DEST[55:48]-SRC[55:48]
DEST[63:56] ←DEST[63:56]-SRC[63:56]
DEST[71:64] ←DEST[71:64]-SRC[71:64]
DEST[79:72] ←DEST[79:72]-SRC[79:72]
DEST[87:80] ←DEST[87:80]-SRC[87:80]
DEST[95:88] ←DEST[95:88]-SRC[95:88]
DEST[103:96] ←DEST[103:96]-SRC[103:96]
DEST[111:104] ←DEST[111:104]-SRC[111:104]
DEST[119:112] ←DEST[119:112]-SRC[119:112]
DEST[127:120] ←DEST[127:120]-SRC[127:120]
DEST[MAX_VL-1:128] (Unmodified)

VPSUBW (VEX.256 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]

DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[143:128] ←SRC1[143:128]-SRC2[143:128]
DEST[159:144] ←SRC1[159:144]-SRC2[159:144]
DEST[175:160] ←SRC1[175:160]-SRC2[175:160]
DEST[191:176] ←SRC1[191:176]-SRC2[191:176]
DEST[207:192] ←SRC1207:192]-SRC2[207:192]
DEST[223:208] ←SRC1[223:208]-SRC2[223:208]
DEST[239:224] ←SRC1[239:224]-SRC2[239:224]
DEST[255:240] ←SRC1[255:240]-SRC2[255:240]
DEST[MAX_VL-1:256] ←0

VPSUBW (VEX.128 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]
DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[MAX_VL-1:128] ←0

PSUBW (128-bit Legacy SSE version)
DEST[15:0] ←DEST[15:0]-SRC[15:0]
DEST[31:16] ←DEST[31:16]-SRC[31:16]
DEST[47:32] ←DEST[47:32]-SRC[47:32]
DEST[63:48] ←DEST[63:48]-SRC[63:48]
DEST[79:64] ←DEST[79:64]-SRC[79:64]
DEST[95:80] ←DEST[95:80]-SRC[95:80]
DEST[111:96] ←DEST[111:96]-SRC[111:96]
DEST[127:112] ←DEST[127:112]-SRC[127:112]
DEST[MAX_VL-1:128] (Unmodified)

VPSUBD (VEX.256 encoded version)
DEST[31:0] ←SRC1[31:0]-SRC2[31:0]
DEST[63:32] ←SRC1[63:32]-SRC2[63:32]
DEST[95:64] ←SRC1[95:64]-SRC2[95:64]
DEST[127:96] ←SRC1[127:96]-SRC2[127:96]
DEST[159:128] ←SRC1[159:128]-SRC2[159:128]
DEST[191:160] ←SRC1[191:160]-SRC2[191:160]
DEST[223:192] ←SRC1[223:192]-SRC2[223:192]
DEST[255:224] ←SRC1[255:224]-SRC2[255:224]
DEST[MAX_VL-1:256] ←0

VPSUBD (VEX.128 encoded version)
DEST[31:0] ←SRC1[31:0]-SRC2[31:0]
DEST[63:32] ←SRC1[63:32]-SRC2[63:32]
DEST[95:64] ←SRC1[95:64]-SRC2[95:64]
DEST[127:96] ←SRC1[127:96]-SRC2[127:96]
DEST[MAX_VL-1:128] ←0

PSUBD (128-bit Legacy SSE version)
DEST[31:0] ←DEST[31:0]-SRC[31:0]
DEST[63:32] ←DEST[63:32]-SRC[63:32]

```
DEST[95:64] ←DEST[95:64]-SRC[95:64]
DEST[127:96] ←DEST[127:96]-SRC[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16( __mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32( __mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32( __mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32( __mmask8 k, __m128i a, __m128i b);
PSUBB:__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB:__m128i _mm_sub_epi8 ( __m128i a, __m128i b)
VPSUBB:__m256i _mm256_sub_epi8 ( __m256i a, __m256i b)
PSUBW:__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW:__m128i _mm_sub_epi16 ( __m128i a, __m128i b)
VPSUBW:__m256i _mm256_sub_epi16 ( __m256i a, __m256i b)
PSUBD:__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD:__m128i _mm_sub_epi32 ( __m128i a, __m128i b)
VPSUBD:__m256i _mm256_sub_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ *mm1, mm2/m64*	RM	V/V	SSE2	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r PSUBQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed quadword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed quadword integers in *ymm3/m256* from *ymm2*.
EVEX.NDS.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (with 64-Bit operands)

```
    DEST[63:0] ← DEST[63:0] − SRC[63:0];
```

PSUBQ (with 128-Bit operands)

```
    DEST[63:0] ← DEST[63:0] − SRC[63:0];
    DEST[127:64] ← DEST[127:64] − SRC[127:64];
```

VPSUBQ (VEX.128 encoded version)

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

VPSUBQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]
DEST[VLMAX-1:256] ← 0
```

VPSUBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0]
                    ELSE DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]
                FI;
        ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64( __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
```

VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ:__m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ:__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ:__m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBQ, see Exceptions Type E4.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /*r*[1] PSUBSB *mm, mm/m64*	RM	V/V	MMX	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /*r* PSUBSB *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
NP 0F E9 /*r*[1] PSUBSW *mm, mm/m64*	RM	V/V	MMX	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /*r* PSUBSW *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed signed byte integers in *xmm3/m128* from packed signed byte integers in *xmm2* and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed signed word integers in *xmm3/m128* from packed signed word integers in *xmm2* and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed signed byte integers in *ymm3/m256* from packed signed byte integers in *ymm2* and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed signed word integers in *ymm3/m256* from packed signed word integers in *ymm2* and saturate results.
EVEX.NDS.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.

EVEX.NDS.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToSignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat subtract operation for 2nd and 7th words *)
    DEST[63:48] ← SaturateToSignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPSUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
```

VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAX_VL-1:256] ←0;
```

VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAX_VL-1:128] ← 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAX_VL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAX_VL-1:256] ← 0;
```

VPSUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAX_VL-1:128] ← 0;
```

PSUBSW (128-bit Legacy SSE Version)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAX_VL-1:128] (Unmodified);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16( __mmask8 k, __m128i a, __m128i b);
PSUBSB:__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB:__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB:__m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW:__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW:__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW:__m256i _mm256_subs_epi16(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /*r*[1] PSUBUSB *mm, mm/m64*	RM	V/V	MMX	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /*r* PSUBUSB *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /*r*[1] PSUBUSW *mm, mm/m64*	RM	V/V	MMX	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /*r* PSUBUSW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in *xmm1* and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed unsigned byte integers in *xmm3/m128* from packed unsigned byte integers in *xmm2* and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed unsigned word integers in *xmm3/m128* from packed unsigned word integers in *xmm2* and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed unsigned byte integers in *ymm3/m256* from packed unsigned byte integers in *ymm2* and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed unsigned word integers in *ymm3/m256* from packed unsigned word integers in *ymm2* and saturate result.
EVEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.

EVEX.NDS.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] − SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] − SRC[63:56];
```

PSUBUSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBUSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
```

VPSUBUSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16;
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
```

VPSUBUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAX_VL-1:256] ← 0;
```

VPSUBUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAX_VL-1:128] ← 0
```

PSUBUSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAX_VL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 encoded version)
```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAX_VL-1:256] ← 0;
```

VPSUBUSW (VEX.128 encoded version)
```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAX_VL-1:128] ← 0
```

PSUBUSW (128-bit Legacy SSE Version)
```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8( __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8( __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8( __mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16( __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16( __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16( __mmask8 k, __m128i a, __m128i b);
PSUBUSB:__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB:__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB:__m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW:__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW:__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW:__m256i _mm256_subs_epu16(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 68 /*r*[1] PUNPCKHBW *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm.*
66 0F 68 /*r* PUNPCKHBW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 69 /*r*[1] PUNPCKHWD *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm.*
66 0F 69 /*r* PUNPCKHWD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 6A /*r*[1] PUNPCKHDQ *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm.*
66 0F 6A /*r* PUNPCKHDQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1.*
66 0F 6D /*r* PUNPCKHQDQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order bytes from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order words from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order doublewords from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6A /r VPUNPCKHDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6D /r VPUNPCKHQDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.NDS.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

EVEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.NDS.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.NDS.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

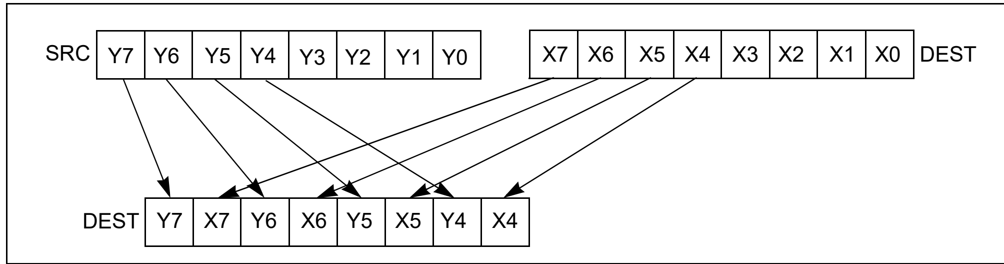


Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

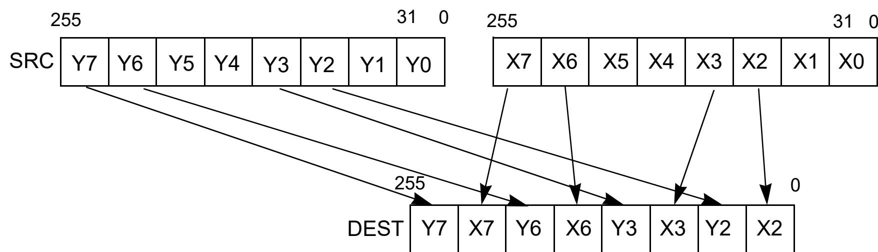


Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW instruction with 64-bit operands:

```
    DEST[7:0] ← DEST[39:32];
    DEST[15:8] ← SRC[39:32];
    DEST[23:16] ← DEST[47:40];
    DEST[31:24] ← SRC[47:40];
    DEST[39:32] ← DEST[55:48];
    DEST[47:40] ← SRC[55:48];
    DEST[55:48] ← DEST[63:56];
    DEST[63:56] ← SRC[63:56];
```

PUNPCKHW instruction with 64-bit operands:

```
    DEST[15:0] ← DEST[47:32];
    DEST[31:16] ← SRC[47:32];
    DEST[47:32] ← DEST[63:48];
    DEST[63:48] ← SRC[63:48];
```

PUNPCKHDQ instruction with 64-bit operands:

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

```
INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
```

```
DEST[167:160] ← SRC1[215:208]
DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[191:184] ← SRC2[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[215:208] ← SRC1[239:232]
DEST[223:216] ← SRC2[239:232]
DEST[231:224] ← SRC1[247:240]
DEST[239:232] ← SRC2[247:240]
DEST[247:240] ← SRC1[255:248]
DEST[255:248] ← SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[191:176] ← SRC2[223:208]
DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
```

DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128] ← SRC1[255:192]
DEST[255:192] ← SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
VPUNPCKHBW (VEX.128 encoded version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[511:127] ←0

VPUNPCKHBW (VEX.256 encoded version)
DEST[255:0] ←INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0

VPUNPCKHBW (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[511:127] ←0
```

VPUNPCKHWD (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKHWD (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
```

```
        ELSE
            IF *merging-masking*            ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[511:127] ←0
```

VPUNPCKHDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKHDQ (EVEX.512 encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*            ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKHQDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKHQDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8( __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8( __mmask16 k, __m128i a, __m128i b);
```

VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW:__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW:__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW:__m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD:__m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD:__m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD:__m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ:__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ:__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ:__m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ:__m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ:__m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 60 /r[1] PUNPCKLBW *mm, mm/m32*	RM	V/V	MMX	Interleave low-order bytes from *mm* and *mm/m32* into *mm.*
66 0F 60 /r PUNPCKLBW *xmm1, xmm2/m128*	RM	V/V	SSE2	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 61 /r[1] PUNPCKLWD *mm, mm/m32*	RM	V/V	MMX	Interleave low-order words from *mm* and *mm/m32* into *mm.*
66 0F 61 /r PUNPCKLWD *xmm1, xmm2/m128*	RM	V/V	SSE2	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 62 /r[1] PUNPCKLDQ *mm, mm/m32*	RM	V/V	MMX	Interleave low-order doublewords from *mm* and *mm/m32* into *mm.*
66 0F 62 /r PUNPCKLDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /r PUNPCKLQDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order bytes from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order words from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order doublewords from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 62 /r VPUNPCKLDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6C /r VPUNPCKLQDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.NDS.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

EVEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

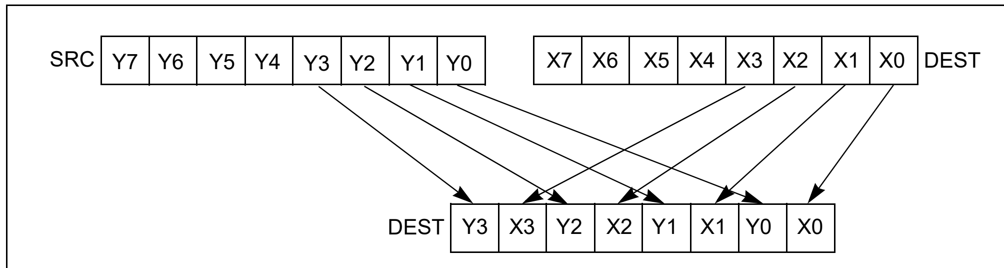


Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

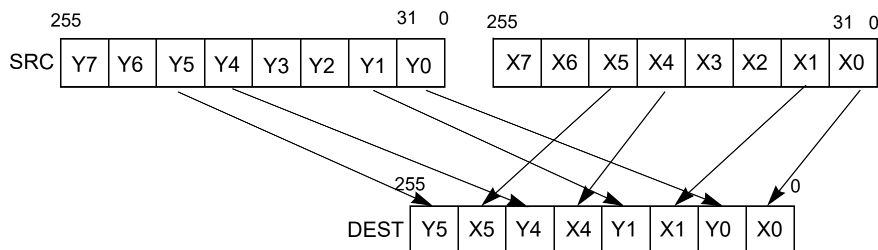


Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

```
PUNPCKLBW instruction with 64-bit operands:
    DEST[63:56] ← SRC[31:24];
    DEST[55:48] ← DEST[31:24];
    DEST[47:40] ← SRC[23:16];
    DEST[39:32] ← DEST[23:16];
    DEST[31:24] ← SRC[15:8];
    DEST[23:16] ← DEST[15:8];
    DEST[15:8] ← SRC[7:0];
    DEST[7:0] ← DEST[7:0];

PUNPCKLWD instruction with 64-bit operands:
    DEST[63:48] ← SRC[31:16];
    DEST[47:32] ← DEST[31:16];
    DEST[31:16] ← SRC[15:0];
    DEST[15:0] ← DEST[15:0];

PUNPCKLDQ instruction with 64-bit operands:
    DEST[63:32] ← SRC[31:0];
    DEST[31:0] ← DEST[31:0];
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
```

```
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ← SRC2[159:152]
DEST[199:192] ← SRC1[167:160]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]
DEST[247:240] ← SRC1[191:184]
DEST[255:248] ← SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
```

DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ←INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ←INTERLEAVE_BYTES(SRC1, SRC2)
DEST[511:127] ←0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ←INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0

VPUNPCKLBW (EVEX.512 encoded instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] ←INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_WORDS(SRC1, SRC2)
DEST[511:127] ←0
```

VPUNPCKLWD (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKLWD (EVEX.512 encoded instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] ←INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKLDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKLDQ (EVEX encoded instructions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

```
ENDFOR
DEST511:0] ←INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAX_VL-1:VL] ← 0
```

PUNPCKLQDQ

```
DEST[127:0] ←INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ←0
```

VPUNPCKLQDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ←0
```

VPUNPCKLQDQ (EVEX encoded instructions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
```

VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);
PUNPCKLBW:__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW:__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW:__m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD:__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD:__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD:__m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ:__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ:__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ:__m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ:__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ:__m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.

PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r[1] PXOR *mm, mm/m64*	RM	V/V	MMX	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /r PXOR *xmm1, xmm2/m128*	RM	V/V	SSE2	Bitwise XOR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EF /r VPXOR *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise XOR of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG EF /r VPXOR *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise XOR of *ymm3/m256* and *ymm2*.
EVEX.NDS.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit operand)

```
DEST ← DEST XOR SRC
```

PXOR (128-bit Legacy SSE version)

```
DEST ← DEST XOR SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPXOR (VEX.128 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[VLMAX-1:128] ← 0
```

VPXOR (VEX.256 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[VLMAX-1:256] ← 0
```

VPXORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VPXORQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32( __mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32( __mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32( __mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64( __m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64( __m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64( __m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 ( __m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 53 /*r* RCPPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 53 /r VRCPPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 53 /r VRCPPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| \leq 1.5 * 2^{-12}$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

RCPPS (128-bit Legacy SSE version)
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[VLMAX-1:128] (Unmodified)

VRCPPS (VEX.128 encoded version)
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[VLMAX-1:128] ← 0

VRCPPS (VEX.256 encoded version)
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS: __m128 _mm_rcp_ps(__m128 a)

RCPPS: __m256 _mm256_rcp_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

RDPID—Read Processor ID

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C7 /7 RDPID r32	R	N.E./V	RDPID	Read IA32_TSC_AUX into r32.
F3 0F C7 /7 RDPID r64	R	V/N.E.	RDPID	Read IA32_TSC_AUX into r64.

Instruction Operand Encoding[1]

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

Description

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

Operation

DEST ← IA32_TSC_AUX

Flags Affected

None.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the F2 prefix is used.
If CPUID.7H.0:ECX.RDPID[bit 22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

1. ModRM.MOD = 011B required

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EE	RDPKRU	ZO	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

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

Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDPKRU: uint32_t _rdpkru_u32(void);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	ZO	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Instruction Operand Encoding

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

Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do no support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for "fast" reads.

The ECX register specifies the counter type (if the processor supports architectural performance monitoring) and counter index. Counter type is specified in ECX[30] to select one of two type of performance counters. If the processor does not support architectural performance monitoring, ECX[30:0] specifies the counter index; otherwise ECX[29:0] specifies the index relative to the base of each counter type. ECX[31] selects "fast" read mode if supported. The two counter types are:

- General-purpose or special-purpose performance counters are specified with ECX[30] = 0: The number of general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 18, "Performance Monitoring" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Table 4-16.
- Fixed-function performance counters are specified with ECX[30] = 1. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 18, "Performance Monitoring" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific.

Table 4-16 lists valid indices of the general-purpose and special-purpose performance counters according to the DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_00H, 0FH_01H, 0FH_02H, 0FH_03H, 0FH_04H, 0FH_06H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17

Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel® Core™2 Duo processor family, Intel Xeon processor 3100, 3300, 5200, 5400 series - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
45 nm and 32 nm Intel® Atom™ processors	06H_1CH, 06_26H, 06_27H, 06_35H, 06_36H	0, 1	0, 1
Intel® Atom™ processors based on Silvermont or Airmont microarchitectures	06H_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_4CH	0, 1	0, 1
Next Generation Intel® Atom™ processors based on Goldmont microarchitecture	06H_5CH, 06_5FH	0-3	0-3
Intel® processors based on the Nehalem, Westmere microarchitectures	06H_1AH, 06H_1EH, 06H_1FH, 06_25H, 06_2CH, 06H_2EH, 06_2FH	0-3	0-3
Intel® processors based on the Sandy Bridge, Ivy Bridge microarchitecture	06H_2AH, 06H_2DH, 06H_3AH, 06H_3EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)
Intel® processors based on the Haswell, Broadwell, SkyLake microarchitectures	06H_3CH, 06H_45H, 06H_46H, 06H_3FH, 06_3DH, 06_47H, 4FH, 06_56H, 06_4EH, 06_5EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)

Processors based on Intel NetBurst microarchitecture support "fast" (32-bit) and "slow" (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on these processors than a full 40-bit read.

On processors based on Intel NetBurst microarchitecture with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDMPC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, "Performance Monitoring Events," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If

an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

```
(* Intel processors that support architectural performance monitoring *)

Most significant counter bit (MSCB) = 47

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

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

Most significant counter bit (MSCB) = 39

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

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

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors based on Intel NetBurst microarchitecture *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
```

```
                THEN
                    EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                    EDX ← PMC(ECX[30:0])[39:32];
            ELSE (* ECX[31] = 1*)
                THEN
                    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                    EDX ← 0;
            FI;
    ELSE IF (*64-bit Intel processor based on Intel NetBurst microarchitecture with L3 *)
            THEN IF (ECX[30:0] = 18:25 )
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
            FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-19. *)
            GP(0);
    FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

RET—Return from Procedure

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm16	NA	NA	NA

Description

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

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

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

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

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

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

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

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

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

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

Operation

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

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

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
```

```
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP ← SP + (SRC AND FFFFH);
    FI;
FI;

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

RETURN-TO-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
```

```
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
    FI;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
            THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
            THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
```

```
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP + SRC;
                            ELSE (* StackAddressSize = 16 *)
                                SP ← SP + SRC;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        FI;

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

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

    (* IA-32e Mode *)
        IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
            THEN
                IF OperandSize = 32
                    THEN
                        IF second doubleword on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second doubleword on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE
                        IF OperandSize = 16
                            THEN
                                IF second word on stack is not within stack limits
                                    THEN #SS(0); FI;
                                IF first or second word on stack is not in canonical space
                                    THEN #SS(0); FI;
                            ELSE (* OperandSize = 64 *)
                                IF first or second quadword on stack is not in canonical space
```

```
                                    THEN #SS(0); FI;
                    FI
                FI;
            IF return code segment selector is NULL
                THEN GP(0); FI;
            IF return code segment selector addresses descriptor beyond descriptor table limit
                THEN GP(selector); FI;
            IF return code segment selector addresses descriptor in non-canonical space
                THEN GP(selector); FI;
            Obtain descriptor to which return code segment selector points from descriptor table;
            IF return code segment descriptor is not a code segment
                THEN #GP(selector); FI;
            IF return code segment descriptor has L-bit = 1 and D-bit = 1
                THEN #GP(selector); FI;
            IF return code segment selector RPL < CPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is conforming
            and return code segment DPL > return code segment selector RPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is non-conforming
            and return code segment DPL ≠ return code segment selector RPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is not present
                THEN #NP(selector); FI:
            IF return code segment selector RPL > CPL
                THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
                ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
            FI;
        FI;

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

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

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

```
                        IF StackAddressSize = 16
                            THEN
                                SP ← SP + SRC;
                            ELSE (* StackAddressSize = 64 *)
                                RSP ← RSP + SRC;
                        FI;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP + SRC;
                            ELSE
                                IF StackAddressSize = 16
                                    THEN
                                        SP ← SP + SRC;
                                    ELSE (* StackAddressSize = 64 *)
                                        RSP ← RSP + SRC;
                                FI;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP ← tempESP;
                SS ← tempSS;
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        RSP ← RSP + SRC;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        FI;
FI;

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

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

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

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
	If the return instruction pointer is not within the return code segment limit.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return code segment selector is NULL.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 52 /r RSQRTPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 52 /r VRSQRTPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 52 /r VRSQRTPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

RSQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[VLMAX-1:128] (Unmodified)
```

VRSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[VLMAX-1:128] ← 0
```

VRSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS: __m128 _mm_rsqrt_ps(__m128 a)

RSQRTPS: __m256 _mm256_rsqrt_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

SFENCE—Store Fence

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F8	SFENCE	ZO	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction. The SFENCE instruction is ordered with respect to store instructions, other SFENCE instructions, any LFENCE and MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_sfence(void)
```

Exceptions (All Operating Modes)

#UD If CPUID.01H:EDX.SSE[bit 25] = 0.
If the LOCK prefix is used.

SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT *m*	M	Valid	Valid	Store GDTR to *m.*

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in bytes 3-5, and byte 6 is zero-filled. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In IA-32e mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize = 16
            THEN
                DEST[0:15] ← GDTR(Limit);
                DEST[16:39] ← GDTR(Base); (* 24 bits of base address stored *)
                DEST[40:47] ← 0;
            ELSE IF (32-bit Operand Size)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
                FI;
            ELSE (* 64-bit Operand Size *)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

Operation

SHA1MSG1

```
W0 ← SRC1[127:96] ;
W1 ← SRC1[95:64] ;
W2 ← SRC1[63: 32] ;
W3 ← SRC1[31: 0] ;
W4 ← SRC2[127:96] ;
W5 ← SRC2[95:64] ;

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1: __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

Operation

SHA1MSG2

```
W13 ← SRC2[95:64] ;
W14 ← SRC2[63: 32] ;
W15 ← SRC2[31: 0] ;
W16 ← (SRC1[127:96] XOR W13 ) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63: 32] XOR W15) ROL 1;
W19 ← (SRC1[31: 0] XOR W16) ROL 1;

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.

Instruction Operand Encoding

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

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

```
TMP ← (SRC1[127:96] ROL 30);

DEST[127:96] ← SRC2[127:96] + TMP;
DEST[95:64] ← SRC2[95:64];
DEST[63:32] ← SRC2[63:32];
DEST[31:0] ← SRC2[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1RNDS4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A CC /r ib SHA1RNDS4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8

Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation

SHA1RNDS4

The function f() and Constant K are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() ← f0(), K ← K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() ← f1(), K ← K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() ← f2(), K ← K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() ← f3(), K ← K3;
FI;

A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];
W0E ← SRC2[127:96];
W1 ← SRC2[95:64];
W2 ← SRC2[63:32];
W3 ← SRC2[31:0];

Round i = 0 operation:
A_1 ← f (B, C, D) + (A ROL 5) +W0E +K;
B_1 ← A;
C_1 ← B ROL 30;
D_1 ← C;
E_1 ← D;

FOR i = 1 to 3
    A_(i +1) ← f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
    B_(i +1) ← A_i;
```

```
    C_(i +1) ← B_i ROL 30;
    D_(i +1) ← C_i;
    E_(i +1) ← D_i;
ENDFOR

DEST[127:96] ← A_4;
DEST[95:64] ← B_4;
DEST[63:32] ← C_4;
DEST[31:0] ← D_4;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1RNDS4: __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

Operation

SHA256MSG1

```
W4 ← SRC2[31: 0] ;
W3 ← SRC1[127:96] ;
W2 ← SRC1[95:64] ;
W1 ← SRC1[63: 32] ;
W0 ← SRC1[31: 0] ;

DEST[127:96] ← W3 + σ0( W4);
DEST[95:64] ← W2 + σ0( W3);
DEST[63:32] ← W1 + σ0( W2);
DEST[31:0] ← W0 + σ0( W1);
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 ← SRC2[95:64] ;
W15 ← SRC2[127:96] ;
W16 ← SRC1[31: 0] + σ_1(W14) ;
W17 ← SRC1[63: 32] + σ_1(W15) ;
W18 ← SRC1[95: 64] + σ_1(W16) ;
W19 ← SRC1[127: 96] + σ_1(W17) ;

DEST[127:96] ← W19 ;
DEST[95:64] ← W18 ;
DEST[63:32] ← W17 ;
DEST[31:0] ← W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256RNDS2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CB /r SHA256RNDS2 xmm1, xmm2/m128, <XMM0>	RM0	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

Description

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

Operation

SHA256RNDS2

```
A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK0 ← XMM0[31: 0];
WK1 ← XMM0[63: 32];

FOR i = 0 to 1
    A_(i +1) ← Ch (E_i, F_i, G_i) +Σ1( E_i) +WKi+ H_i + Maj(A_i , B_i, C_i) +Σ0( A_i);
    B_(i +1) ← A_i;
    C_(i +1) ← B_i ;
    D_(i +1) ← C_i;
    E_(i +1) ← Ch (E_i, F_i, G_i) +Σ1( E_i) +WKi+ H_i + D_i;
    F_(i +1) ← E_i ;
    G_(i +1) ← F_i;
    H_(i +1) ← G_i;
ENDFOR

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDS2: __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	RMI	V/V	SSE	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.NDS.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.NDS.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.NDS.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

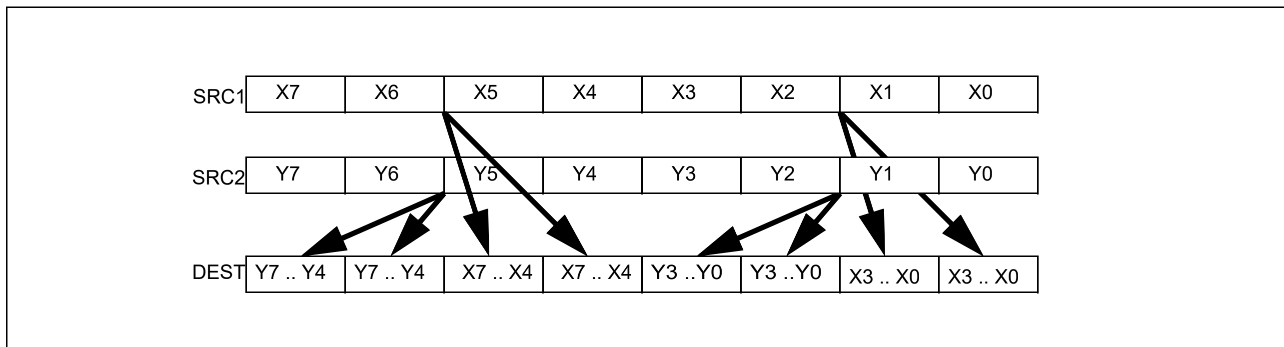


Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:   TMP ←SRC[31:0];
    1:   TMP ←SRC[63:32];
    2:   TMP ←SRC[95:64];
    3:   TMP ←SRC[127:96];
ESAC;
RETURN TMP
}
```

VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VPSHUFPS (EVEX encoded versions when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 0
```

VSHUFPS (VEX.256 encoded version)

```
DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ←Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ←Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ←Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ←Select4(SRC2[255:128], imm8[7:6]);
DEST[MAX_VL-1:256] ←0
```

VSHUFPS (VEX.128 encoded version)

```
DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] ←0
```

SHUFPS (128-bit Legacy SSE version)

```
DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT *m*	M	Valid	Valid	Store IDTR to *m*.

Instruction Operand Encoding

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

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, if the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize = 16
            THEN
                DEST[0:15] ← IDTR(Limit);
                DEST[16:39] ← IDTR(Base); (* 24 bits of base address stored; *)
                DEST[40:47] ← 0;
            ELSE IF (32-bit Operand Size)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
            ELSE (* 64-bit Operand Size *)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	RM	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← SQRT(SRC[31:0])
                ELSE DEST[i+31:i] ← SQRT(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VSQRTPS (VEX.256 encoded version)

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

VSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[MAX_VL-1:128] ←0
```

SQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.NDS.LIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAX_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VSQRTSD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

SQRTSD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

SQRTSS—Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

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

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX_VL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI;
FI;
DEST[127:31] ← SRC1[127:31]
DEST[MAX_VL-1:128] ← 0
```

VSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

SQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss( __mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CB STAC	ZO	V/V	SMAP	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

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

Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

Operation

```
EFLAGS.AC ← 1;
```

Flags Affected

AC set. Other flags are unaffected.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /3 STMXCSR *m32*	M	V/V	SSE	Store contents of MXCSR register to *m32*.
VEX.LZ.0F.WIG AE /3 VSTMXCSR *m32*	M	V/V	AVX	Store contents of MXCSR register to *m32*.

Instruction Operand Encoding

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

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

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

VEX.L must be 0, otherwise instructions will #UD.

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

Operation

```
m32 ← MXCSR;
```

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

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

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	RM	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.NDS.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VSUBPS (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0];
                ELSE    DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];
            FI;

    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

VSUBPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC2[159:128]
DEST[191:160]← SRC1[191:160] - SRC2[191:160]
DEST[223:192] ← SRC1[223:192] - SRC2[223:192]
DEST[255:224] ← SRC1[255:224] - SRC2[255:224].
DEST[MAX_VL-1:256] ← 0
```

VSUBPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[MAX_VL-1:128] ← 0
```

SUBPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

SUBSD—Subtract Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	RM	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

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

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

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

VSUBSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] - SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAX_VL-1:128] ←0
```

SUBSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] - SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

SUBSS—Subtract Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	RM	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

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

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

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

VSUBSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] - SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAX_VL-1:128] ←0
```

SUBSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] - SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISS (all versions)

```
RESULT ←    UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUCOMISS  int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS   int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomineq_ss(__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

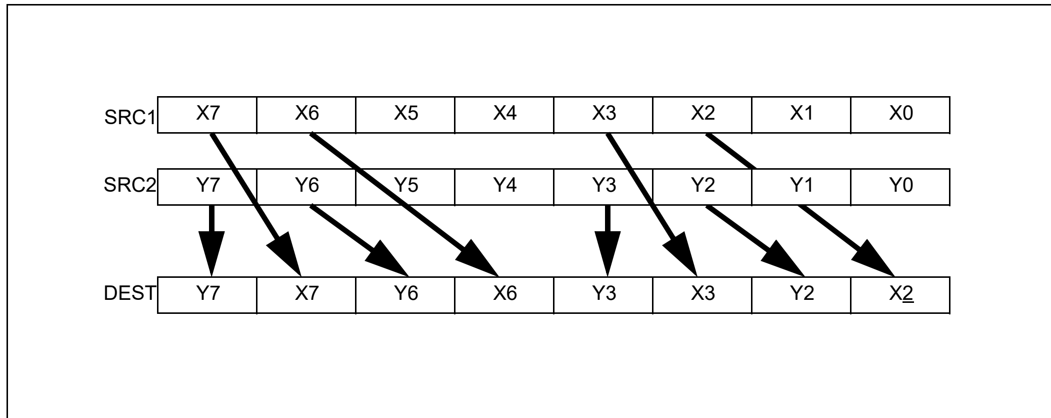


Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX encoded version when SRC2 is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← SRC2[511:480]
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VUNPCKHPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← TMP_SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← TMP_SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← TMP_SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← TMP_SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← TMP_SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VUNPCKHPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[159:128] ←SRC1[223:192]
DEST[191:160] ←SRC2[223:192]
DEST[223:192] ←SRC1[255:224]
DEST[255:224] ←SRC2[255:224]
DEST[MAX_VL-1:256] ← 0
```

VUNPCKHPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[MAX_VL-1:128] ←0
```

UNPCKHPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.NDS.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.NDS.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

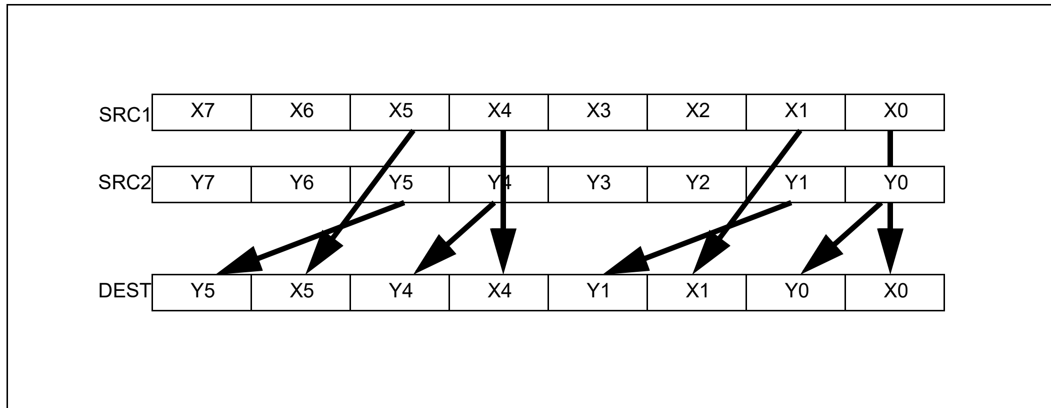


Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[31:0]
    TMP_DEST[63:32] ← SRC2[31:0]
    TMP_DEST[95:64] ← SRC1[63:32]
    TMP_DEST[127:96] ← SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*               ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VUNPCKLPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← TMP_SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← TMP_SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← TMP_SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← TMP_SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← TMP_SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← TMP_SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*               ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

```
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

UNPCKLPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[159:128] ←SRC1[159:128]
DEST[191:160] ←SRC2[159:128]
DEST[223:192] ←SRC1[191:160]
DEST[255:224] ←SRC2[191:160]
DEST[MAX_VL-1:256] ← 0
```

VUNPCKLPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[MAX_VL-1:128] ←0
```

UNPCKLPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

6. Updates to Chapter 5, Volume 2C

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C:* Instruction Set Reference, V-Z.

Change to this chapter: Updates to the following instructions: VRANGEPD, VRANGEPS, WRPKRU, XEND, XGETBV, XORPS, XRSTOR/XRSTOR64, XRSTORS/XRSTORS64, XSAVE/XSAVE64, XSAVEC/XSAVEC64, XSAVEOPT/XSAVEOPT64, XSAVES/XSAVES64, XSETBV and XTEST.

VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

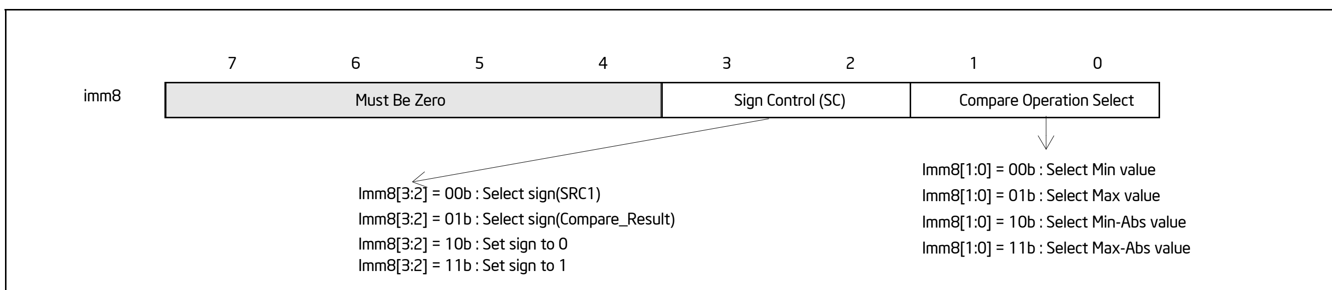


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2] has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Table 5-12. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]

Src1	Src2	Result	IE Signaling Due to Comparison	Imm8[3:2] Effect to Range Output
sNaN1	sNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

Table 5-13. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS and MAX, MAX_ABS

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

Table 5-14. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)

MIN_ABS (|a| = |b|, a>0, b<0)			MAX_ABS (|a| = |b|, a>0, b<0)		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
         IF DAZ THEN Src1.fraction ← 0;
         ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
         IF DAZ THEN Src2.fraction ← 0;
         ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
    ELSE
         Case(CmpOpCtl[1:0])
         00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
         01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
         10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
         11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
         ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

zmm_dst is the destination operand.

zmm_src is the input operand to compare against ±1023 (this is SRC1).

zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d _mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd (__m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512DQ	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
         IF DAZ THEN Src1.fraction ← 0;
         ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
         IF DAZ THEN Src2.fraction ← 0;
         ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF     (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
    ELSE
         Case(CmpOpCtl[1:0])
         00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
         01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
         10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
         11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
         ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b == 1) AND (SRC2 *is memory*)
                    THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                    ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
                FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:

zmm_dst is the destination operand.

zmm_src is the input operand to compare against ±150.

zmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512 _mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

WRPKRU—Write Data to User Page Key Register

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF	WRPKRU	ZO	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

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

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU: void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
	If EDX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XEND — Transactional End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND: void _xend(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK or 66H or F2H or F3H prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D0	XGETBV	ZO	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

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

Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, "Managing State Using the XSAVE Feature Set," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

```
EDX:EAX ← XCR[ECX];
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV: unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX_VL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

VXORPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].
DEST[MAX_VL-1:256] ← 0
```

VXORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAX_VL-1:128] ← 0
```

XORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
```

```
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /5	XRSTOR *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F AE /5	XRSTOR64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.8, "Operation of XRSTOR," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
```

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

```
RSTORMASK ← XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED ← RFBM AND RSTORMASK;
        TO_BE_INITIALIZED ← RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
        FI;

        IF RFBM[1] = 1 OR RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
                XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0; // this step does not initialize MXCSR
                XINUSE[1] ← 0;
        FI;

        FOR i ← 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset n from base of XSAVE area;
                        // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
                    XINUSE[i] ← 1;
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN    /* compacted form not supported */
                #GP(0);
        FI;

        FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
        RESTORE_FEATURES = FORMAT AND RFBM;
```

```
        TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
        FORCE_INIT ← RFBM AND NOT FORMAT;
        TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
                XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0;
                MXCSR ← 1F80H;
                XINUSE[1] ← 0;
        FI;

        NEXT_FEATURE_OFFSET = 576;              // Legacy area and XSAVE header consume 576 bytes
        FOR i ← 2 TO 62
            IF FORMAT[i] = 1
                THEN
                    IF TO_BE_RESTORED[i] = 1
                        THEN
                            load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                            XINUSE[i] ← 1;
                    FI;
                    NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
            FI;
        ENDFOR;
FI;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void _xrstor(void * , unsigned __int64);

XRSTOR: void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /3	XRSTORS *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F C7 /3	XRSTORS64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.12, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;            /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
```

```
FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        initialize x87 state;
        XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        MXCSR ← 1F80H;
        XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
    FI;
ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← 〈CPL,VMXNR,LAXA,COMPMASK〉;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: void _xrstors(void * , unsigned __int64);

XRSTORS64: void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /3	XRSTORS *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F C7 /3	XRSTORS64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.12, "Operation of XRSTORS," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area").
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;            /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
```

```
FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        initialize x87 state;
        XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        MXCSR ← 1F80H;
        XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
    FI;
ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: void _xrstors(void * , unsigned __int64);

XRSTORS64: void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVE—Save Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /4	XSAVE *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem*.
NP REX.W + 0F AE /4	XSAVE64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.7, "Operation of XSAVE," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

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

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
```

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

```
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE: void _xsave(void * , unsigned __int64);

XSAVE: void _xsave64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /4	XSAVEC *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction.
NP REX.W + 0F C7 /4	XSAVEC64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.10, "Operation of XSAVEC," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

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

Operation

```
RFBM ← XCR0 AND EDX:EAX;                /* bitwise logical AND */
TO_BE_SAVED ← RFBM AND XINUSE;          /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
```

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

```
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC: void _xsavec(void * , unsigned __int64);

XSAVEC64: void _xsavec64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /6 XSAVEOPT *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.
NP REX.W + 0F AE /6 XSAVEOPT64 *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

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

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
```

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

```
OLD_BV ← XSTATE_BV field from XSAVE header;
TO_BE_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,00000000_00000000H⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void _xsaveopt(void * , unsigned __int64);

XSAVEOPT: void _xsaveopt64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /5	XSAVES *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.
NP REX.W + 0F C7 /5	XSAVES64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.11, "Operation of XSAVES," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

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

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
TO_BE_SAVED ← RFBM AND XINUSE;
IF XRSTOR_INFO = 〈CPL,VMXNR,LAXA,COMPMASK〉
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;            // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8          // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] ← 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES: void _xsaves(void * , unsigned __int64);

XSAVES64: void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	IIf the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an align-

ment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSETBV—Set Extended Control Register

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

Instruction Operand Encoding

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

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

```
XCR[ECX] ← EDX:EAX;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV: void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int _xtest(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.
	If LOCK prefix is used.

7. Updates to Chapter 1, Volume 3A

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

Change to this chapter: Added reference to new volume 4: Model Specific Registers.

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1* (order number 253668), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2* (order number 253669), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*: *System Programming Guide, Part 3* (order number 326019), and the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4* (order number 332831) are part of a set that describes the architecture and programming environment of Intel 64 and IA-32 Architecture processors. The other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*: *Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

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

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

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

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

- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes.
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows[1]:

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

Chapter 2 — System Architecture Overview. Describes the modes of operation used by Intel 64 and IA-32 processors and the mechanisms provided by the architectures to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

Chapter 3 — Protected-Mode Memory Management. Describes the data structures, registers, and instructions that support segmentation and paging. The chapter explains how they can be used to implement a "flat" (unsegmented) memory model or a segmented memory model.

Chapter 4 — Paging. Describes the paging modes supported by Intel 64 and IA-32 processors.

Chapter 5 — Protection. Describes the support for page and segment protection provided in the Intel 64 and IA-32 architectures. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes.

Chapter 6 — Interrupt and Exception Handling. Describes the basic interrupt mechanisms defined in the Intel 64 and IA-32 architectures, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each exception is given in this chapter. Includes programming the LINT0 and LINT1 inputs and gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

Chapter 7 — Task Management. Describes mechanisms the Intel 64 and IA-32 architectures provide to support multitasking and inter-task protection.

Chapter 8 — Multiple-Processor Management. Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and Intel® Hyper-Threading Technology. Includes MP initialization for P6 family processors and gives an example of how to use the MP protocol to boot P6 family processors in an MP system.

Chapter 9 — Processor Management and Initialization. Defines the state of an Intel 64 or IA-32 processor after reset initialization. This chapter also explains how to set up an Intel 64 or IA-32 processor for real-address mode operation and protected- mode operation, and how to switch between modes.

Chapter 10 — Advanced Programmable Interrupt Controller (APIC). Describes the programming interface to the local APIC and gives an overview of the interface between the local APIC and the I/O APIC. Includes APIC bus message formats and describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

Chapter 11 — Memory Cache Control. Describes the general concept of caching and the caching mechanisms supported by the Intel 64 or IA-32 architectures. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. Information on using the new cache control and memory streaming instructions introduced with the Pentium III, Pentium 4, and Intel Xeon processors is also given.

Chapter 12 — Intel® MMX™ Technology System Programming. Describes those aspects of the Intel® MMX™ technology that must be handled and considered at the system programming level, including: task switching, exception handling, and compatibility with existing system environments.

Chapter 13 — System Programming For Instruction Set Extensions And Processor Extended States. Describes the operating system requirements to support SSE/SSE2/SSE3/SSSE3/SSE4 extensions, including task switching, exception handling, and compatibility with existing system environments. The latter part of this chapter describes the extensible framework of operating system requirements to support processor extended states. Processor extended state may be required by instruction set extensions beyond those of SSE/SSE2/SSE3/SSSE3/SSE4 extensions.

1. Model-Specific Registers have been moved out of this volume and into a separate volume: *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

Chapter 14 — Power and Thermal Management. Describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

Chapter 15 — Machine-Check Architecture. Describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, and P6 family processors. Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

Chapter 16 — Interpreting Machine-Check Error Codes. Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

Chapter 17 — Debug, Branch Profile, TSC, and Resource Monitoring Features. Describes the debugging registers and other debug mechanism provided in Intel 64 or IA-32 processors. This chapter also describes the time-stamp counter.

Chapter 18 — Performance Monitoring. Describes the Intel 64 and IA-32 architectures' facilities for monitoring performance.

Chapter 19 — Performance-Monitoring Events. Lists architectural performance events. Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture.

Chapter 20 — 8086 Emulation. Describes the real-address and virtual-8086 modes of the IA-32 architecture.

Chapter 21 — Mixing 16-Bit and 32-Bit Code. Describes how to mix 16-bit and 32-bit code modules within the same program or task.

Chapter 22 — IA-32 Architecture Compatibility. Describes architectural compatibility among IA-32 processors.

Chapter 23 — Introduction to Virtual Machine Extensions. Describes the basic elements of virtual machine architecture and the virtual machine extensions for Intel 64 and IA-32 Architectures.

Chapter 24 — Virtual Machine Control Structures. Describes components that manage VMX operation. These include the working-VMCS pointer and the controlling-VMCS pointer.

Chapter 25 — VMX Non-Root Operation. Describes the operation of a VMX non-root operation. Processor operation in VMX non-root mode can be restricted programmatically such that certain operations, events or conditions can cause the processor to transfer control from the guest (running in VMX non-root mode) to the monitor software (running in VMX root mode).

Chapter 26 — VM Entries. Describes VM entries. VM entry transitions the processor from the VMM running in VMX root-mode to a VM running in VMX non-root mode. VM-Entry is performed by the execution of VMLAUNCH or VMRESUME instructions.

Chapter 27 — VM Exits. Describes VM exits. Certain events, operations or situations while the processor is in VMX non-root operation may cause VM-exit transitions. In addition, VM exits can also occur on failed VM entries.

Chapter 28 — VMX Support for Address Translation. Describes virtual-machine extensions that support address translation and the virtualization of physical memory.

Chapter 29 — APIC Virtualization and Virtual Interrupts. Describes the VMCS including controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

Chapter 30 — VMX Instruction Reference. Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

Chapter 31 — Virtual-Machine Monitor Programming Considerations. Describes programming considerations for VMMs. VMMs manage virtual machines (VMs).

Chapter 32 — Virtualization of System Resources. Describes the virtualization of the system resources. These include: debugging facilities, address translation, physical memory, and microcode update facilities.

Chapter 33 — Handling Boundary Conditions in a Virtual Machine Monitor. Describes what a VMM must consider when handling exceptions, interrupts, error conditions, and transitions between activity states.

Chapter 34 — System Management Mode. Describes Intel 64 and IA-32 architectures' system management mode (SMM) facilities.

Chapter 35 — Intel® Processor Trace. Describes details of Intel® Processor Trace.

Chapter 36 — Introduction to Intel® Software Guard Extensions. Provides an overview of the Intel® Software Guard Extensions (Intel® SGX) set of instructions.

Chapter 37 — Enclave Access Control and Data Structures. Describes Enclave Access Control procedures and defines various Intel SGX data structures.

Chapter 38 — Enclave Operation. Describes enclave creation and initialization, adding pages and measuring an enclave, and enclave entry and exit.

Chapter 39 — Enclave Exiting Events. Describes enclave-exiting events (EEE) and asynchronous enclave exit (AEX).

Chapter 40 — SGX Instruction References. Describes the supervisor and user level instructions provided by Intel SGX.

Chapter 41 — Intel® SGX Interactions with IA32 and Intel® 64 Architecture. Describes the Intel SGX collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel 64 architectures.

Chapter 42 — Enclave Code Debug and Profiling. Describes enclave code debug processes and options.

Appendix A — VMX Capability Reporting Facility. Describes the VMX capability MSRs. Support for specific VMX features is determined by reading capability MSRs.

Appendix B — Field Encoding in VMCS. Enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

Appendix C — VM Basic Exit Reasons. Describes the 32-bit fields that encode reasons for a VM exit. Examples of exit reasons include, but are not limited to: software interrupts, processor exceptions, software traps, NMIs, external interrupts, and triple faults.

1.3 NOTATIONAL CONVENTIONS

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

1.3.1 Bit and Byte Order

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

1.3.2 Reserved Bits and Software Compatibility

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

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

NOTE

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

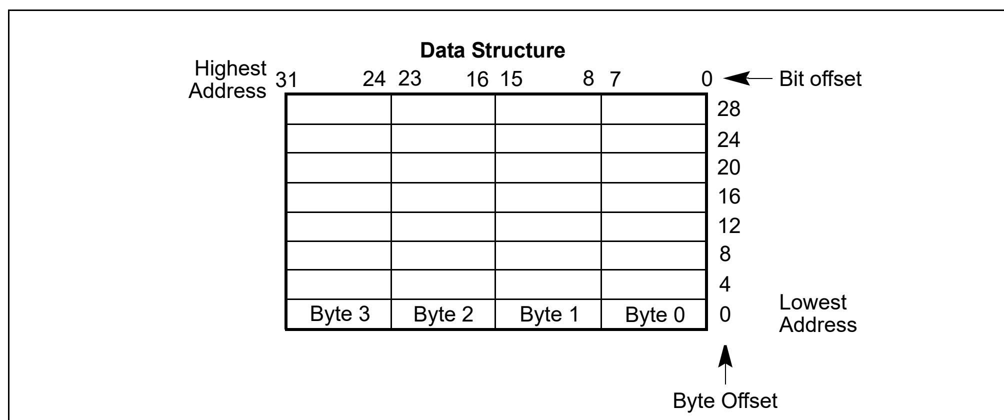


Figure 1-1. Bit and Byte Order

1.3.3 Instruction Operands

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

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

where:

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

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

For example:

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

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

1.3.4 Hexadecimal and Binary Numbers

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

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

1.3.5 Segmented Addressing

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

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

Segment-register:Byte-address

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

DS:FF79H

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

CS:EIP

1.3.6 Syntax for CPUID, CR, and MSR Values

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

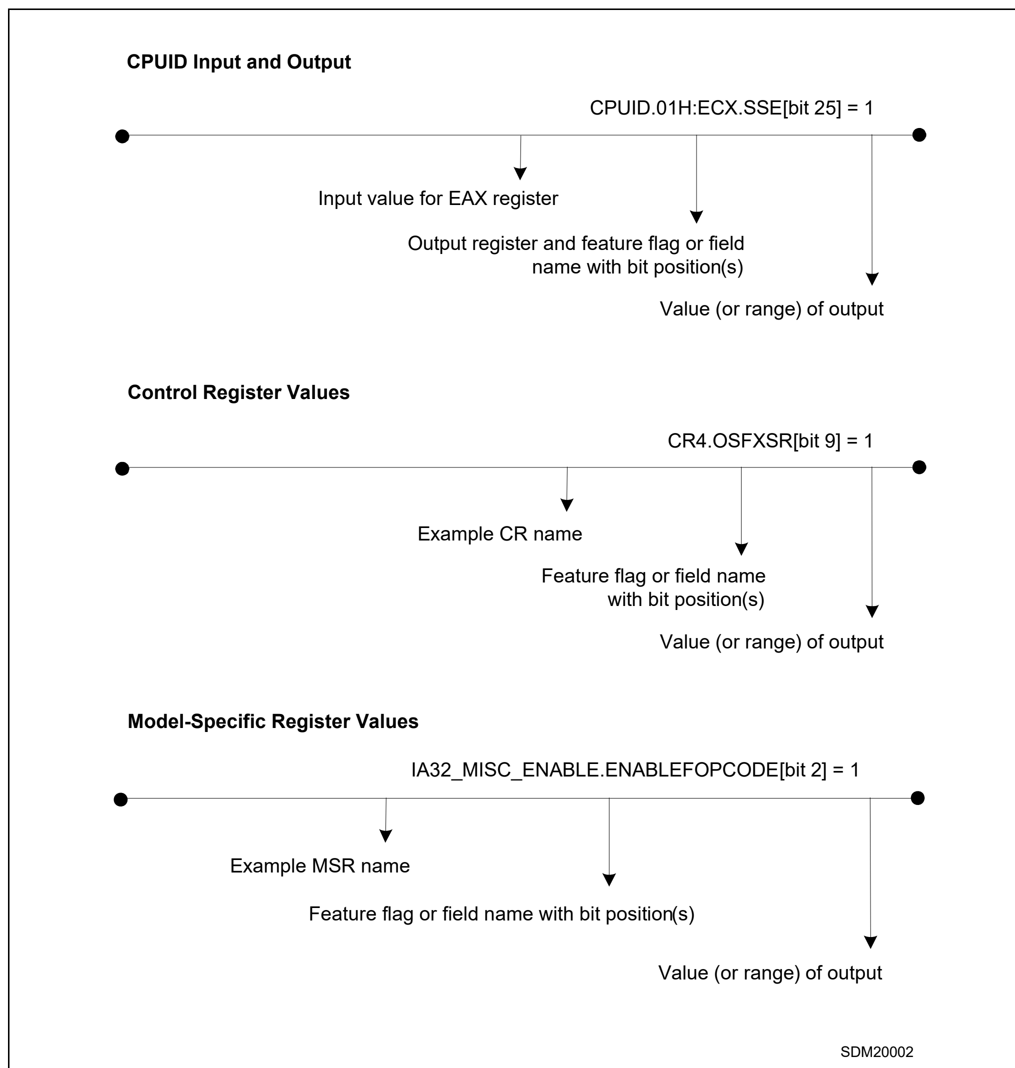


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.7 Exceptions

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

#PF(fault code)

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

#GP(0)

1.4 RELATED LITERATURE

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

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

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- Intel 64 Architecture x2APIC Specification:

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

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/
- Performance Monitoring Unit Sharing Guide
http://software.intel.com/file/30388

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

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

More relevant links are:

- Intel® Developer Zone:

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

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

  http://www.intel.com/support/processors/
- Software products and packages:

  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® Hyper-Threading Technology (Intel® HT Technology):

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

8. Updates to Chapter 2, Volume 3A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Change to this chapter: Added reference to new volume 4: Model Specific Registers. Updated term “IA-32e paging” to “4-level paging”; included footnote on first usage of new term.

CHAPTER 2
SYSTEM ARCHITECTURE OVERVIEW

IA-32 architecture (beginning with the Intel386 processor family) provides extensive support for operating-system and system-development software. This support offers multiple modes of operation, which include:

- Real mode, protected mode, virtual 8086 mode, and system management mode. These are sometimes referred to as legacy modes.

Intel 64 architecture supports almost all the system programming facilities available in IA-32 architecture and extends them to a new operating mode (IA-32e mode) that supports a 64-bit programming environment. IA-32e mode allows software to operate in one of two sub-modes:

- 64-bit mode supports 64-bit OS and 64-bit applications
- Compatibility mode allows most legacy software to run; it co-exists with 64-bit applications under a 64-bit OS.

The IA-32 system-level architecture includes features to assist in the following operations:

- Memory management
- Protection of software modules
- Multitasking
- Exception and interrupt handling
- Multiprocessing
- Cache management
- Hardware resource and power management
- Debugging and performance monitoring

This chapter provides a description of each part of this architecture. It also describes the system registers that are used to set up and control the processor at the system level and gives a brief overview of the processor's system-level (operating system) instructions.

Many features of the system-level architecture are used only by system programmers. However, application programmers may need to read this chapter and the following chapters in order to create a reliable and secure environment for application programs.

This overview and most subsequent chapters of this book focus on protected-mode operation of the IA-32 architecture. IA-32e mode operation of the Intel 64 architecture, as it differs from protected mode operation, is also described.

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset (see Chapter 9, "Processor Management and Initialization"). Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE

System-level architecture consists of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt and exception handling, task management, and control of multiple processors.

Figure 2-1 provides a summary of system registers and data structures that applies to 32-bit modes. System registers and data structures that apply to IA-32e mode are shown in Figure 2-2.

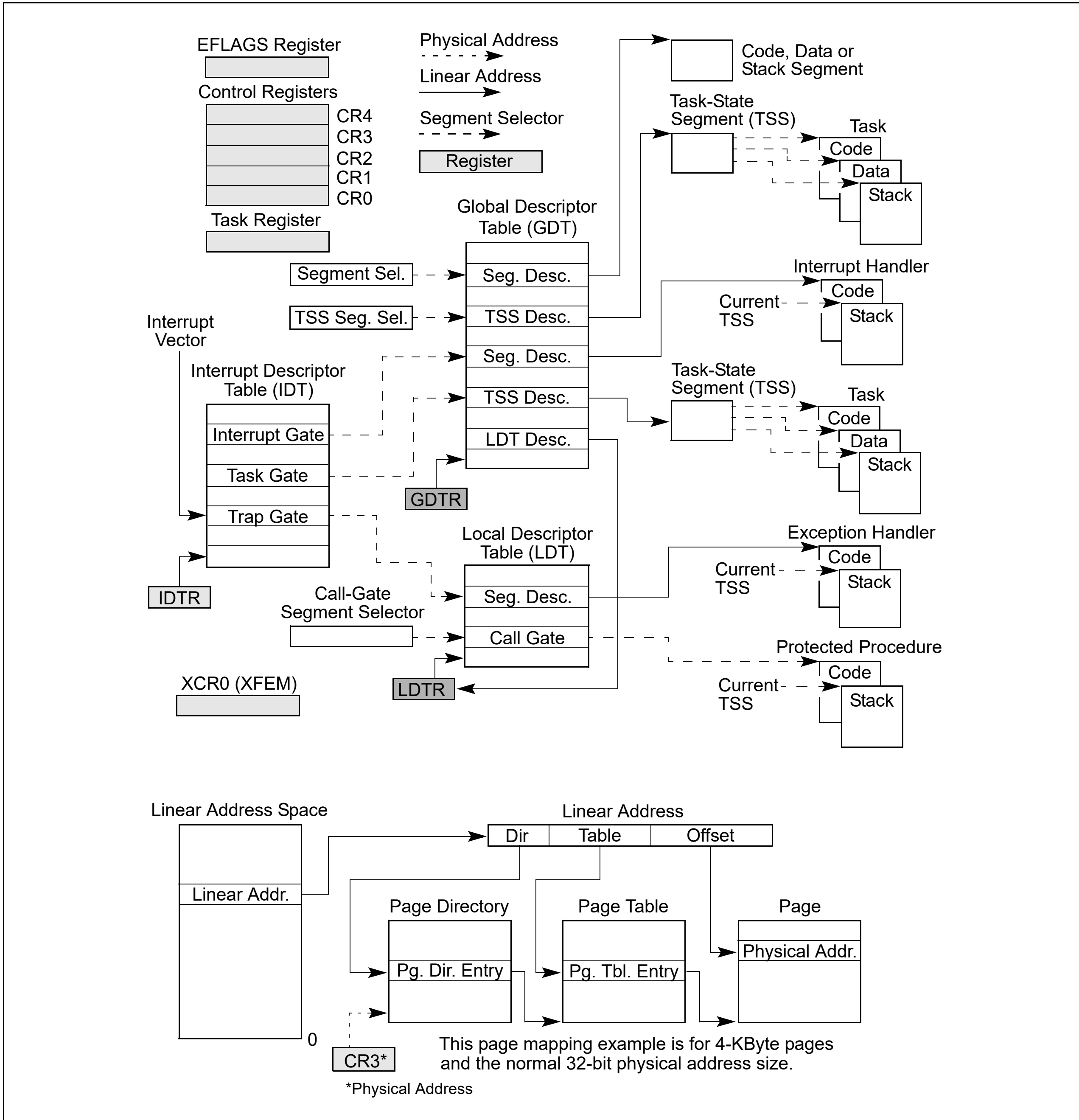


Figure 2-1. IA-32 System-Level Registers and Data Structures

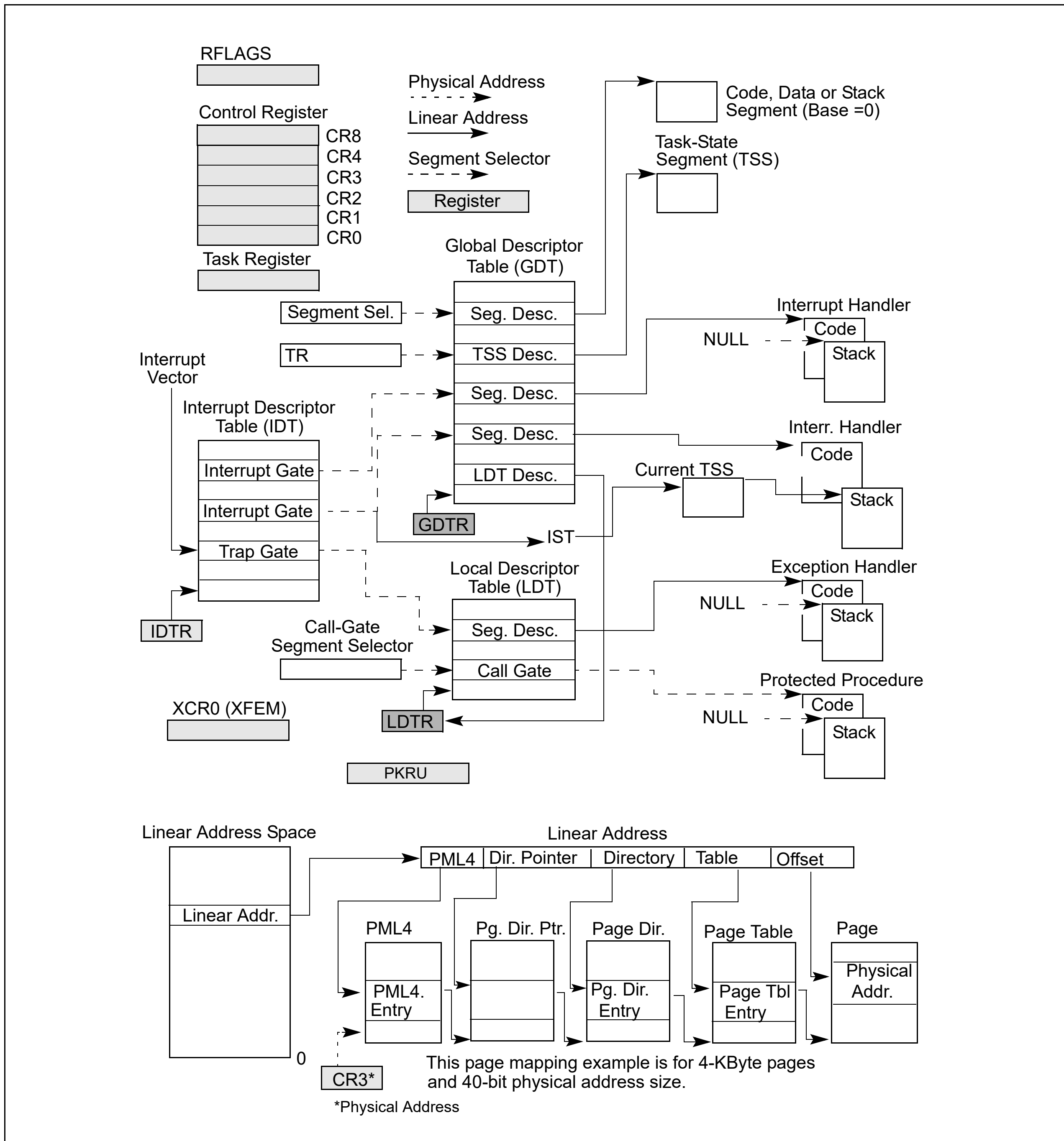


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

See Figure 2-1. The solid arrows in the figure indicate a linear address, dashed lines indicate a segment selector, and the dotted arrows indicate a physical address. For simplicity, many of the segment selectors are shown as direct pointers to a segment. However, the actual path from a segment selector to its associated segment is always through a GDT or LDT.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode

GDTR and LDTR registers are expanded to 64-bits wide in both IA-32e sub-modes (64-bit mode and compatibility mode). For more information: see Section 3.5.2, “Segment Descriptor Tables in IA-32e Mode.”

Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses, (16-byte LDT descriptors hold a 64-bit base address and various attributes). In compatibility mode, descriptors are not expanded.

2.1.2 System Segments, Segment Descriptors, and Gates

Besides code, data, and stack segments that make up the execution environment of a program or procedure, the architecture defines two system segments: the task-state segment (TSS) and the LDT. The GDT is not considered a segment because it is not accessed by means of a segment selector and segment descriptor. TSSs and LDTs have segment descriptors defined for them.

The architecture also defines a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures. For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level (more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure[1] supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate.

If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

2.1.2.1 Gates in IA-32e Mode

In IA-32e mode, the following descriptors are 16-byte descriptors (expanded to allow a 64-bit base): LDT descriptors, 64-bit TSSs, call gates, interrupt gates, and trap gates.

Call gates facilitate transitions between 64-bit mode and compatibility mode. Task gates are not supported in IA-32e mode. On privilege level changes, stack segment selectors are not read from the TSS. Instead, they are set to NULL.

---

1. The word “procedure” is commonly used in this document as a general term for a logical unit or block of code (such as a program, procedure, function, or routine).

2.1.3 Task-State Segments and Task Gates

The TSS (see Figure 2-1) defines the state of the execution environment for a task. It includes the state of general-purpose registers, segment registers, the EFLAGS register, the EIP register, and segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging-structure hierarchy.

All program execution in protected mode happens within the context of a task (called the current task). The segment selector for the TSS for the current task is stored in the task register. The simplest method for switching to a task is to make a call or jump to the new task. Here, the segment selector for the TSS of the new task is given in the CALL or JMP instruction. In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.
2. Loads the task register with the segment selector for the new task.
3. Accesses the new TSS through a segment descriptor in the GDT.
4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.
5. Begins execution of the new task.

A task can also be accessed through a task gate. A task gate is similar to a call gate, except that it provides access (through a segment selector) to a TSS rather than a code segment.

2.1.3.1 Task-State Segments in IA-32e Mode

Hardware task switches are not supported in IA-32e mode. However, TSSs continue to exist. The base address of a TSS is specified by its descriptor.

A 64-bit TSS holds the following information that is important to 64-bit operation:

- Stack pointer addresses for each privilege level
- Pointer addresses for the interrupt stack table
- Offset address of the IO-permission bitmap (from the TSS base)

The task register is expanded to hold 64-bit base addresses in IA-32e mode. See also: Section 7.7, "Task Management in 64-bit Mode."

2.1.4 Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software by means of an INT, INTO, INT 3, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

2.1.4.1 Interrupt and Exception Handling IA-32e Mode

In IA-32e mode, interrupt gate descriptors are expanded to 16 bytes to support 64-bit base addresses. This is true for 64-bit mode and compatibility mode.

The IDTR register is expanded to hold a 64-bit base address. Task gates are not supported.

2.1.5 Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory.

The location of pages (sometimes called page frames) in physical memory is contained in the paging structures. These structures reside in physical memory (see Figure 2-1 for the case of 32-bit paging).

The base physical address of the paging-structure hierarchy is contained in control register CR3. The entries in the paging structures determine the physical address of the base of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into parts. The parts provide separate offsets into the paging structures and the page frame. A system can have a single hierarchy of paging structures or several. For example, each task can have its own hierarchy.

2.1.5.1 Memory Management in IA-32e Mode

In IA-32e mode, physical memory pages are managed by a set of system data structures. In compatibility mode and 64-bit mode, four levels of system data structures are used. These include:

- **The page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. The base physical address of the PML4 is stored in CR3.
- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.
- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.
- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

2.1.6 System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights. See also: Section 2.3, “System Flags and Fields in the EFLAGS Register.”
- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive. See also: Section 2.5, “Control Registers” and Section 2.6, “Extended Control Registers (Including XCR0).”
- The debug registers (not shown in Figure 2-1) allow the setting of breakpoints for use in debugging programs and systems software. See also: Chapter 17, “Debug, Branch Profile, TSC, and Resource Monitoring Features.”
- The GDTR, LDTR, and IDTR registers contain the linear addresses and sizes (limits) of their respective tables. See also: Section 2.4, “Memory-Management Registers.”
- The task register contains the linear address and size of the TSS for the current task. See also: Section 2.4, “Memory-Management Registers.”
- Model-specific registers (not shown in Figure 2-1).

The model-specific registers (MSRs) are a group of registers available primarily to operating-system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine- check architecture, and the memory type ranges (MTRRs).

The number and function of these registers varies among different members of the Intel 64 and IA-32 processor families. See also: Section 9.4, “Model-Specific Registers (MSRs),” and Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

Most systems restrict access to system registers (other than the EFLAGS register) by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

2.1.6.1 System Registers in IA-32e Mode

In IA-32e mode, the four system-descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded in hardware to hold 64-bit base addresses. EFLAGS becomes the 64-bit RFLAGS register. CR0–CR4 are expanded to 64 bits. CR8 becomes available. CR8 provides read-write access to the task priority register (TPR) so that the operating system can control the priority classes of external interrupts.

In 64-bit mode, debug registers DR0–DR7 are 64 bits. In compatibility mode, address-matching in DR0–DR3 is also done at 64-bit granularity.

On systems that support IA-32e mode, the extended feature enable register (IA32_EFER) is available. This model-specific register controls activation of IA-32e mode and other IA-32e mode operations. In addition, there are several model-specific registers that govern IA-32e mode instructions:

- **IA32_KERNEL_GS_BASE** — Used by SWAPGS instruction.
- **IA32_LSTAR** — Used by SYSCALL instruction.
- **IA32_FMASK** — Used by SYSCALL instruction.
- **IA32_STAR** — Used by SYSCALL and SYSRET instruction.

2.1.7 Other System Resources

Besides the system registers and data structures described in the previous sections, system architecture provides the following additional resources:

- Operating system instructions (see also: Section 2.8, “System Instruction Summary”).
- Performance-monitoring counters (not shown in Figure 2-1).
- Internal caches and buffers (not shown in Figure 2-1).

Performance-monitoring counters are event counters that can be programmed to count processor events such as the number of instructions decoded, the number of interrupts received, or the number of cache loads. See also: Chapter 19, “Performance Monitoring Events.”

The processor provides several internal caches and buffers. The caches are used to store both data and instructions. The buffers are used to store things like decoded addresses to system and application segments and write operations waiting to be performed. See also: Chapter 11, “Memory Cache Control.”

2.2 MODES OF OPERATION

The IA-32 architecture supports three operating modes and one quasi-operating mode:

- **Protected mode** — This is the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to existing software base.
- **Real-address mode** — This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).
- **System management mode (SMM)** — SMM is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.
- **Virtual-8086 mode** — In protected mode, the processor supports a quasi-operating mode known as virtual-8086 mode. This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

- **IA-32e mode** — In IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 2-3 shows how the processor moves between operating modes.

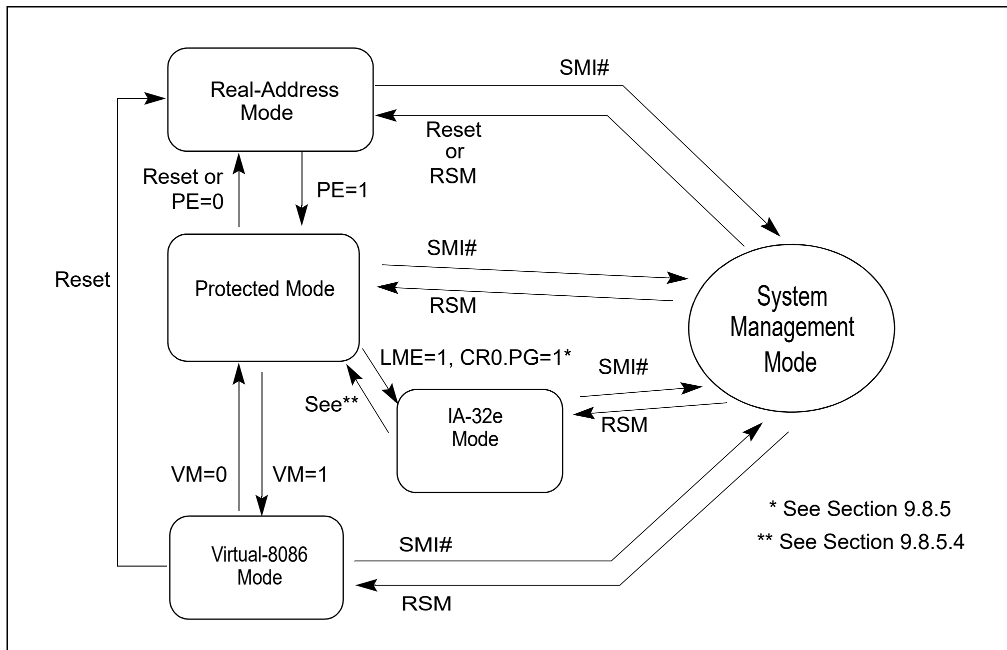


Figure 2-3. Transitions Among the Processor's Operating Modes

The processor is placed in real-address mode following power-up or a reset. The PE flag in control register CR0 then controls whether the processor is operating in real-address or protected mode. See also: Section 9.9, "Mode Switching." and Section 4.1.2, "Paging-Mode Enabling."

The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. See also: Section 20.2.5, "Entering Virtual-8086 Mode."

The LMA bit (IA32_EFER.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32_EFER.LME[bit 8]). See also: Chapter 9, "Processor Management and Initialization."

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

2.2.1 Extended Feature Enable Register

The IA32_EFER MSR provides several fields related to IA-32e mode enabling and operation. It also provides one field that relates to page-access right modification (see Section 4.6, "Access Rights"). The layout of the IA32_EFER MSR is shown in Figure 2-4.

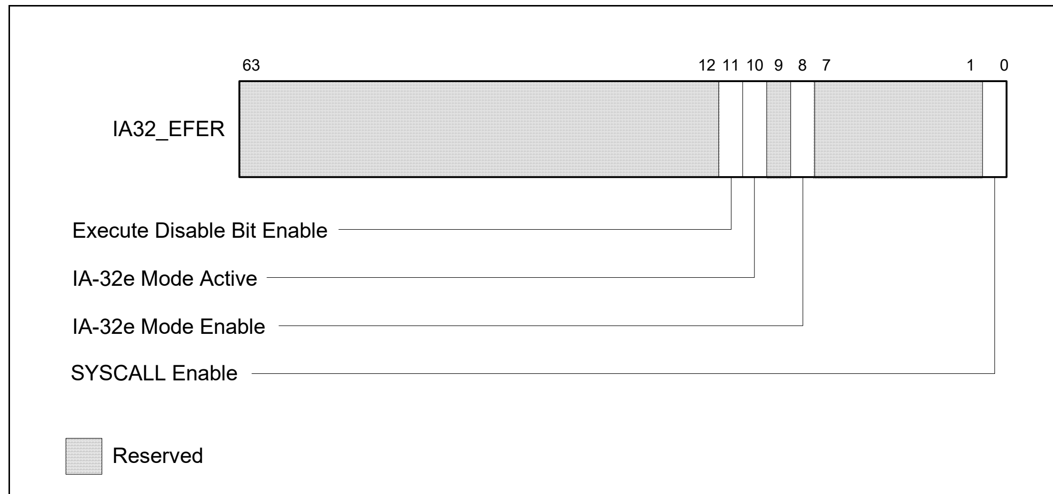


Figure 2-4. IA32_EFER MSR Layout

Table 2-1. IA32_EFER MSR Information

Bit	Description
0	**SYSCALL Enable: IA32_EFER.SCE (R/W)** Enables SYSCALL/SYSRET instructions in 64-bit mode.
7:1	Reserved.
8	**IA-32e Mode Enable: IA32_EFER.LME (R/W)** Enables IA-32e mode operation.
9	Reserved.
10	**IA-32e Mode Active: IA32_EFER.LMA (R)** Indicates IA-32e mode is active when set.
11	**Execute Disable Bit Enable: IA32_EFER.NXE (R/W)** Enables page access restriction by preventing instruction fetches from PAE pages with the XD bit set (See Section 4.6).
63:12	Reserved.

2.3 SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER

The system flags and IOPL field of the EFLAGS register control I/O, maskable hardware interrupts, debugging, task switching, and the virtual-8086 mode (see Figure 2-5). Only privileged code (typically operating system or executive code) should be allowed to modify these bits.

The system flags and IOPL are:

TF **Trap (bit 8)** — Set to enable single-step mode for debugging; clear to disable single-step mode. In single-step mode, the processor generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If an application program sets the TF flag using a

POPF, POPFD, or IRET instruction, a debug exception is generated after the instruction that follows the POPF, POPFD, or IRET.

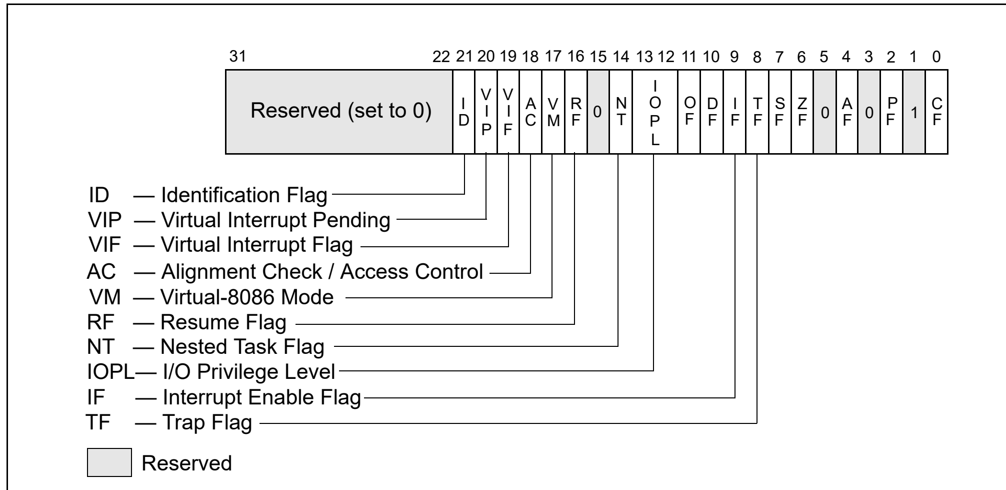


Figure 2-5. System Flags in the EFLAGS Register

IF **Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 6.3.2, "Maskable Hardware Interrupts"). The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.

IOPL **I/O privilege level field (bits 12 and 13)** — Indicates the I/O privilege level (IOPL) of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

The IOPL is also one of the mechanisms that controls the modification of the IF flag and the handling of interrupts in virtual-8086 mode when virtual mode extensions are in effect (when CR4.VME = 1). See also: Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

NT **Nested task (bit 14)** — Controls the chaining of interrupted and called tasks. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. It examines and modifies this flag on returns from a task initiated with the IRET instruction. The flag can be explicitly set or cleared with the POPF/POPFD instructions; however, changing to the state of this flag can generate unexpected exceptions in application programs.

See also: Section 7.4, "Task Linking."

RF **Resume (bit 16)** — Controls the processor's response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints (although other exception conditions can cause an exception to be generated). When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debug software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with IRETD (to prevent the instruction breakpoint from causing another debug exception). The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See also: Section 17.3.1.1, "Instruction-Breakpoint Exception Condition."

VM **Virtual-8086 mode (bit 17)** — Set to enable virtual-8086 mode; clear to return to protected mode.

See also: Section 20.2.1, “Enabling Virtual-8086 Mode.”

AC **Alignment check or access control (bit 18)** — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. An alignment-check exception is generated when reference is made to an unaligned operand, such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. Alignment-check exceptions are generated only in user mode (privilege level 3). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate this exception even when caused by instructions executed in user-mode.

The alignment-check exception can be used to check alignment of data. This is useful when exchanging data with processors which require all data to be aligned. The alignment-check exception can also be used by interpreters to flag some pointers as special by misaligning the pointer. This eliminates overhead of checking each pointer and only handles the special pointer when used.

If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1. See Section 4.6, “Access Rights.”

VIF **Virtual Interrupt (bit 19)** — Contains a virtual image of the IF flag. This flag is used in conjunction with the VIP flag. The processor only recognizes the VIF flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. (The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.)

See also: Section 20.3.3.5, “Method 6: Software Interrupt Handling,” and Section 20.4, “Protected-Mode Virtual Interrupts.”

VIP **Virtual interrupt pending (bit 20)** — Set by software to indicate that an interrupt is pending; cleared to indicate that no interrupt is pending. This flag is used in conjunction with the VIF flag. The processor reads this flag but never modifies it. The processor only recognizes the VIP flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.

See Section 20.3.3.5, “Method 6: Software Interrupt Handling,” and Section 20.4, “Protected-Mode Virtual Interrupts.”

ID **Identification (bit 21)** — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

2.3.1 System Flags and Fields in IA-32e Mode

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. System flags in RFLAGS (64-bit mode) or EFLAGS (compatibility mode) are shown in Figure 2-5.

In IA-32e mode, the processor does not allow the VM bit to be set because virtual-8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit. The processor does, however, allow software to set the NT bit (note that an IRET causes a general protection fault in IA-32e mode if the NT bit is set).

In IA-32e mode, the SYSCALL/SYSRET instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save/restore EFLAGS/RFLAGS.

2.4 MEMORY-MANAGEMENT REGISTERS

The processor provides four memory-management registers (GDTR, LDTR, IDTR, and TR) that specify the locations of the data structures which control segmented memory management (see Figure 2-6). Special instructions are provided for loading and storing these registers.

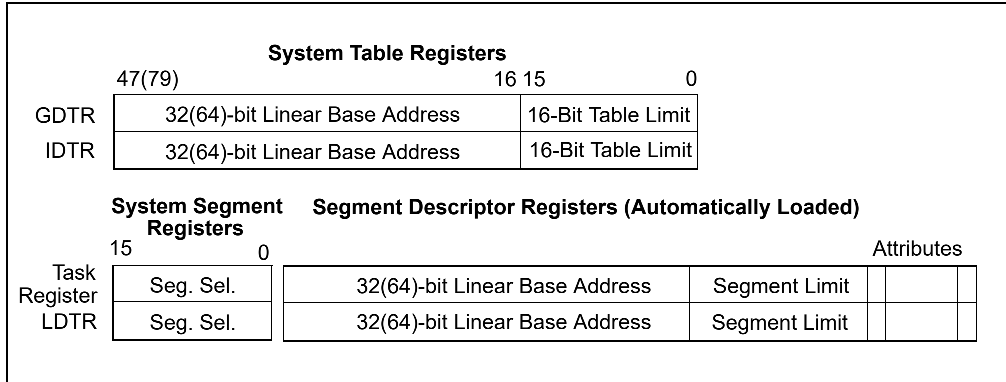


Figure 2-6. Memory Management Registers

2.4.1 Global Descriptor Table Register (GDTR)

The GDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and the 16-bit table limit for the GDT. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table.

The LGDT and SGDT instructions load and store the GDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. A new base address must be loaded into the GDTR as part of the processor initialization process for protected-mode operation.

See also: Section 3.5.1, "Segment Descriptor Tables."

2.4.2 Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the LDT. The base address specifies the linear address of byte 0 of the LDT segment; the segment limit specifies the number of bytes in the segment. See also: Section 3.5.1, "Segment Descriptor Tables."

The LLDT and SLDT instructions load and store the segment selector part of the LDTR register, respectively. The segment that contains the LDT must have a segment descriptor in the GDT. When the LLDT instruction loads a segment selector in the LDTR: the base address, limit, and descriptor attributes from the LDT descriptor are automatically loaded in the LDTR.

When a task switch occurs, the LDTR is automatically loaded with the segment selector and descriptor for the LDT for the new task. The contents of the LDTR are not automatically saved prior to writing the new LDT information into the register.

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

2.4.3 IDTR Interrupt Descriptor Table Register

The IDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and 16-bit table limit for the IDT. The base address specifies the linear address of byte 0 of the IDT; the table limit specifies the number of bytes in the table. The LIDT and SIDT instructions load and store the IDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. The base address and limit in the register can then be changed as part of the processor initialization process.

See also: Section 6.10, "Interrupt Descriptor Table (IDT)."

2.4.4 Task Register (TR)

The task register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the TSS of the current task. The selector references the TSS descriptor in the GDT. The base address specifies the linear address of byte 0 of the TSS; the segment limit specifies the number of bytes in the TSS. See also: Section 7.2.4, "Task Register."

The LTR and STR instructions load and store the segment selector part of the task register, respectively. When the LTR instruction loads a segment selector in the task register, the base address, limit, and descriptor attributes from the TSS descriptor are automatically loaded into the task register. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH.

When a task switch occurs, the task register is automatically loaded with the segment selector and descriptor for the TSS for the new task. The contents of the task register are not automatically saved prior to writing the new TSS information into the register.

2.5 CONTROL REGISTERS

Control registers (CR0, CR1, CR2, CR3, and CR4; see Figure 2-7) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode.

In 64-bit mode, control registers are expanded to 64 bits. The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored. The following is also true:

- The control registers can be read and loaded (or modified) using the move-to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating-system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.
- Bits 63:32 of CR0 and CR4 are reserved and must be written with zeros. Writing a nonzero value to any of the upper 32 bits results in a general-protection exception, #GP(0).
- All 64 bits of CR2 are writable by software.
- Bits 51:40 of CR3 are reserved and must be 0.
- The MOV CRn instructions do not check that addresses written to CR2 and CR3 are within the linear-address or physical-address limitations of the implementation.
- Register CR8 is available in 64-bit mode only.

The control registers are summarized below, and each architecturally defined control field in these control registers is described individually. In Figure 2-7, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0).

- **CR0** — Contains system control flags that control operating mode and states of the processor.
- **CR1** — Reserved.
- **CR2** — Contains the page-fault linear address (the linear address that caused a page fault).
- **CR3** — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor's internal data caches (they do not control TLB caching of page-directory information).

  When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table. In IA-32e mode, the CR3 register contains the base address of the PML4 table.

  See also: Chapter 4, "Paging."
- **CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities.

- **CR8** — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

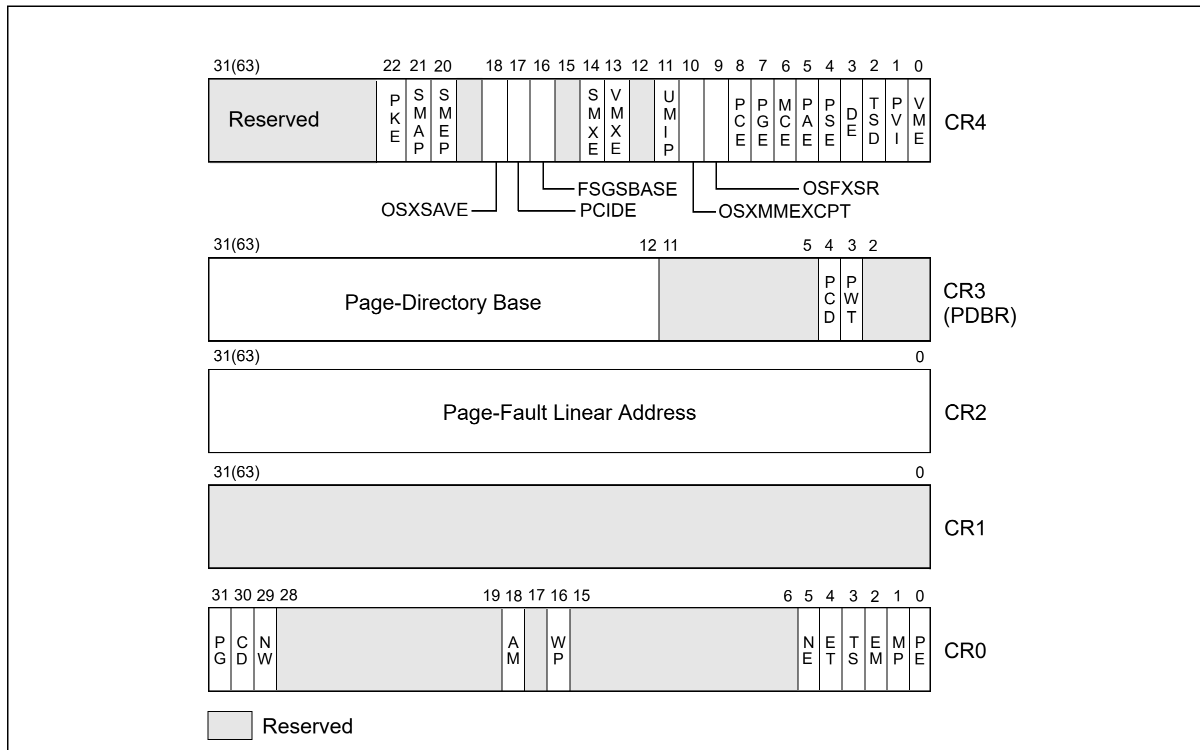


Figure 2-7. Control Registers

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

PG **Paging (bit 31 of CR0)** — Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, "Paging."

On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

CD **Cache Disable (bit 30 of CR0)** — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor's internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur.

See also: Section 11.5.3, "Preventing Caching," and Section 11.5, "Cache Control."

NW **Not Write-through (bit 29 of CR0)** — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

AM **Alignment Mask (bit 18 of CR0)** — Enables automatic alignment checking when set; disables alignment checking when clear. Alignment checking is performed only when the AM flag is set, the AC flag in the EFLAGS register is set, CPL is 3, and the processor is operating in either protected or virtual-8086 mode.

WP **Write Protect (bit 16 of CR0)** — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.

NE **Numeric Error (bit 5 of CR0)** — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.

The FERR# pin is intended to drive an input to an external interrupt controller (the FERR# pin emulates the ERROR# pin of the Intel 287 and Intel 387 DX math coprocessors). The NE flag, IGNNE# pin, and FERR# pin are used with external logic to implement PC-style error reporting. Using FERR# and IGNNE# to handle floating-point exceptions is deprecated by modern operating systems; this non-native approach also limits newer processors to operate with one logical processor active.

See also: Section 8.7, "Handling x87 FPU Exceptions in Software" in Chapter 8, "Programming with the x87 FPU," and Appendix A, "EFLAGS Cross-Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

ET **Extension Type (bit 4 of CR0)** — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

TS **Task Switched (bit 3 of CR0)** — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

- If the TS flag is set and the EM flag (bit 2 of CR0) is clear, a device-not-available exception (#NM) is raised prior to the execution of any x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction; with the exception of PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. See the paragraph below for the special case of the WAIT/FWAIT instructions.
- If the TS flag is set and the MP flag (bit 1 of CR0) and EM flag are clear, an #NM exception is not raised prior to the execution of an x87 FPU WAIT/FWAIT instruction.
- If the EM flag is set, the setting of the TS flag has no effect on the execution of x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

Table 2-2 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 12-1 and 13-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction.

The processor does not automatically save the context of the x87 FPU, XMM, and MXCSR registers on a task switch. Instead, it sets the TS flag, which causes the processor to raise an #NM exception whenever it encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction in the instruction stream for the new task (with the exception of the instructions listed above).

The fault handler for the #NM exception can then be used to clear the TS flag (with the CLTS instruction) and save the context of the x87 FPU, XMM, and MXCSR registers. If the task never encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction, the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context is never saved.

Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS

CR0 Flags			x87 FPU Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute.
0	0	1	#NM Exception	Execute.
0	1	0	Execute	Execute.
0	1	1	#NM Exception	#NM exception.
1	0	0	#NM Exception	Execute.
1	0	1	#NM Exception	Execute.
1	1	0	#NM Exception	Execute.
1	1	1	#NM Exception	#NM exception.

EM **Emulation (bit 2 of CR0)** — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

When the EM flag is set, execution of an x87 FPU instruction generates a device-not-available exception (#NM). This flag must be set when the processor does not have an internal x87 FPU or is not connected to an external math coprocessor. Setting this flag forces all floating-point instructions to be handled by software emulation. Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the EM, MP, and TS flags.

Also, when the EM flag is set, execution of an MMX instruction causes an invalid-opcode exception (#UD) to be generated (see Table 12-1). Thus, if an IA-32 or Intel 64 processor incorporates MMX technology, the EM flag must be set to 0 to enable execution of MMX instructions.

Similarly for SSE/SSE2/SSE3/SSSE3/SSE4 extensions, when the EM flag is set, execution of most SSE/SSE2/SSE3/SSSE3/SSE4 instructions causes an invalid opcode exception (#UD) to be generated (see Table 13-1). If an IA-32 or Intel 64 processor incorporates the SSE/SSE2/SSE3/SSSE3/SSE4 extensions, the EM flag must be set to 0 to enable execution of these extensions. SSE/SSE2/SSE3/SSSE3/SSE4 instructions not affected by the EM flag include: PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

MP **Monitor Coprocessor (bit 1 of CR0)** — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CR0). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag. Table 9-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the MP, EM, and TS flags.

PE **Protection Enable (bit 0 of CR0)** — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.

See also: Section 9.9, "Mode Switching."

PCD **Page-level Cache Disable (bit 4 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, "Paging and Memory Typing". This bit is not used if paging is disabled, with PAE paging, or with 4-level paging[2] if CR4.PCIDE=1.

PWT **Page-level Write-Through (bit 3 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, "Paging and Memory Typing". This bit is not used if paging is disabled, with PAE paging, or with 4-level paging if CR4.PCIDE=1.

VME **Virtual-8086 Mode Extensions (bit 0 of CR4)** — Enables interrupt- and exception-handling extensions in virtual-8086 mode when set; disables the extensions when clear. Use of the virtual mode extensions can improve the performance of virtual-8086 applications by eliminating the overhead of calling the virtual-

2. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

8086 monitor to handle interrupts and exceptions that occur while executing an 8086 program and, instead, redirecting the interrupts and exceptions back to the 8086 program's handlers. It also provides hardware support for a virtual interrupt flag (VIF) to improve reliability of running 8086 programs in multi-tasking and multiple-processor environments.

See also: Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode."

PVI **Protected-Mode Virtual Interrupts (bit 1 of CR4)** — Enables hardware support for a virtual interrupt flag (VIF) in protected mode when set; disables the VIF flag in protected mode when clear.

See also: Section 20.4, "Protected-Mode Virtual Interrupts."

TSD **Time Stamp Disable (bit 2 of CR4)** — Restricts the execution of the RDTSC instruction to procedures running at privilege level 0 when set; allows RDTSC instruction to be executed at any privilege level when clear. This bit also applies to the RDTSCP instruction if supported (if CPUID.80000001H:EDX[27] = 1).

DE **Debugging Extensions (bit 3 of CR4)** — References to debug registers DR4 and DR5 cause an undefined opcode (#UD) exception to be generated when set; when clear, processor aliases references to registers DR4 and DR5 for compatibility with software written to run on earlier IA-32 processors.

See also: Section 17.2.2, "Debug Registers DR4 and DR5."

PSE **Page Size Extensions (bit 4 of CR4)** — Enables 4-MByte pages with 32-bit paging when set; restricts 32-bit paging to pages of 4 KBytes when clear.

See also: Section 4.3, "32-Bit Paging."

PAE **Physical Address Extension (bit 5 of CR4)** — When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.

See also: Chapter 4, "Paging."

MCE **Machine-Check Enable (bit 6 of CR4)** — Enables the machine-check exception when set; disables the machine-check exception when clear.

See also: Chapter 15, "Machine-Check Architecture."

PGE **Page Global Enable (bit 7 of CR4)** — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory or page-table entry). Global pages are not flushed from the translation-lookaside buffer (TLB) on a task switch or a write to register CR3.

When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.

See also: Section 4.10, "Caching Translation Information."

PCE **Performance-Monitoring Counter Enable (bit 8 of CR4)** — Enables execution of the RDPMC instruction for programs or procedures running at any protection level when set; RDPMC instruction can be executed only at protection level 0 when clear.

OSFXSR
Operating System Support for FXSAVE and FXRSTOR instructions (bit 9 of CR4) — When set, this flag: (1) indicates to software that the operating system supports the use of the FXSAVE and FXRSTOR instructions, (2) enables the FXSAVE and FXRSTOR instructions to save and restore the contents of the XMM and MXCSR registers along with the contents of the x87 FPU and MMX registers, and (3) enables the processor to execute SSE/SSE2/SSE3/SSSE3/SSE4 instructions, with the exception of the PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX registers, but they may not save and restore the contents of the XMM and MXCSR registers. Also, the processor will generate an invalid opcode exception (#UD) if it attempts to execute any SSE/SSE2/SSE3 instruction, with the exception of PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. The operating system or executive must explicitly set this flag.

NOTE

CPUID feature flag FXSR indicates availability of the FXSAVE/FXRSTOR instructions. The OSFXSR bit provides operating system software with a means of enabling FXSAVE/FXRSTOR to save/restore the contents of the X87 FPU, XMM and MXCSR registers. Consequently OSFXSR bit indicates that the operating system provides context switch support for SSE/SSE2/SSE3/SSSE3/SSE4.

OSXMMEXCPT

Operating System Support for Unmasked SIMD Floating-Point Exceptions (bit 10 of CR4) — When set, indicates that the operating system supports the handling of unmasked SIMD floating-point exceptions through an exception handler that is invoked when a SIMD floating-point exception (#XM) is generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3/SSE4.1 SIMD floating-point instructions.

The operating system or executive must explicitly set this flag. If this flag is not set, the processor will generate an invalid opcode exception (#UD) whenever it detects an unmasked SIMD floating-point exception.

UMIP

User-Mode Instruction Prevention (bit 11 of CR4) — When set, the following instructions cannot be executed if CPL > 0: SGDT, SIDT, SLDT, SMSW, and STR. An attempt at such execution causes a general-protection exception (#GP).

VMXE

VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set. See Chapter 23, “Introduction to Virtual Machine Extensions.”

SMXE

SMX-Enable Bit (bit 14 of CR4) — Enables SMX operation when set. See Chapter 6, “Safer Mode Extensions Reference” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.

FSGSBASE

FSGSBASE-Enable Bit (bit 16 of CR4) — Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.

PCIDE

PCID-Enable Bit (bit 17 of CR4) — Enables process-context identifiers (PCIDs) when set. See Section 4.10.1, “Process-Context Identifiers (PCIDs)”. Can be set only in IA-32e mode (if IA32_EFER.LMA = 1).

OSXSAVE

XSAVE and Processor Extended States-Enable Bit (bit 18 of CR4) — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the operating system supports the use of the XGETBV, XSAVE and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 13, “System Programming for Instruction Set Extensions and Processor Extended States”.

SMEP

SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, “Access Rights”.

SMAP

SMAP-Enable Bit (bit 21 of CR4) — Enables supervisor-mode access prevention (SMAP) when set. See Section 4.6, “Access Rights.”

PKE

Protection-Key-Enable Bit (bit 22 of CR4) — Enables 4-level paging to associate each linear address with a protection key. The PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written. This bit also enables access to the PKRU register using the RDPKRU and WRPKRU instructions.

TPL

Task Priority Level (bit 3:0 of CR8) — This sets the threshold value corresponding to the highest-priority interrupt to be blocked. A value of 0 means all interrupts are enabled. This field is available in 64-bit mode. A value of 15 means all interrupts will be disabled.

2.5.1 CPUID Qualification of Control Register Flags

Not all flags in control register CR4 are implemented on all processors. With the exception of the PCE flag, they can be qualified with the CPUID instruction to determine if they are implemented on the processor before they are used.

The CR8 register is available on processors that support Intel 64 architecture.

2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0)

If CPUID.01H:ECX.XSAVE[bit 26] is 1, the processor supports one or more **extended control registers** (XCRs). Currently, the only such register defined is XCR0. This register specifies the set of processor state components for which the operating system provides context management, e.g. x87 FPU state, SSE state, AVX state. The OS programs XCR0 to reflect the features for which it provides context management.

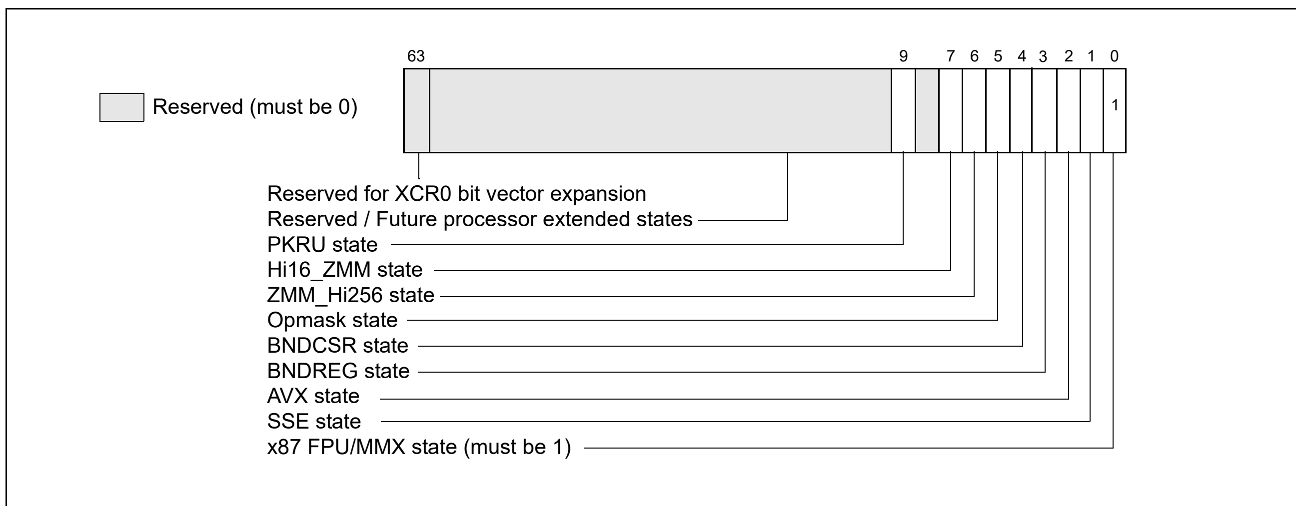


Figure 2-8. XCR0

Software can access XCR0 only if CR4.OSXSAVE[bit 18] = 1. (This bit is also readable as CPUID.01H:ECX.OSXSAVE[bit 27].) Software can use CPUID leaf function 0DH to enumerate the bits in XCR0 that the processor supports (see CPUID instruction in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Each supported state component is represented by a bit in XCR0. System software enables state components by loading an appropriate bit mask value into XCR0 using the XSETBV instruction.

As each bit in XCR0 (except bit 63) corresponds to a processor state component, XCR0 thus provides support for up to 63 sets of processor state components. Bit 63 of XCR0 is reserved for future expansion and will not represent a processor state component.

Currently, XCR0 defines support for the following state components:

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

- XCR0.BNDREG (bit 3): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the bounds registers BND0–BND3.
- XCR0.BNDCSR (bit 4): If 1, MPX instructions can be executed and the XSAVE feature set can be used to manage the BNDCFGU and BNDSTATUS registers.
- XCR0.opmask (bit 5): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the opmask registers k0–k7.
- XCR0.ZMM_Hi256 (bit 6): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the lower ZMM registers (ZMM0-ZMM15 in 64-bit mode; otherwise ZMM0-ZMM7).
- XCR0.Hi16_ZMM (bit 7): If 1, AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper ZMM registers (ZMM16-ZMM31, only in 64-bit mode).
- XCR0.PKRU (bit 9): If 1, the XSAVE feature set can be used to manage the PKRU register (see Section 2.7).

An attempt to use XSETBV to write to XCR0 results in general-protection exceptions (#GP) if it would do any of the following:

- Set a bit reserved in XCR0 for a given processor (as determined by the contents of EAX and EDX after executing CPUID with EAX=0DH, ECX= 0H).
- Clear XCR0.x87.
- Clear XCR0.SSE and set XCR0.AVX.
- Clear XCR0.AVX and set any of XCR0.opmask, XCR0.ZMM_Hi256, and XCR0.Hi16_ZMM.
- Set either XCR0.BNDREG and XCR0.BNDCSR while not setting the other.
- Set any of XCR0.opmask, XCR0.ZMM_Hi256, and XCR0.Hi16_ZMM while not setting all of them.

After reset, all bits (except bit 0) in XCR0 are cleared to zero; XCR0[0] is set to 1.

2.7 PROTECTION KEY RIGHTS REGISTER (PKRU)

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for 4-level paging. The feature allows selective protection of user-mode pages depending on the 4-bit protection key assigned to each page. The **protection key rights register for user pages** (PKRU) allows software to specify the access rights for each protection key.

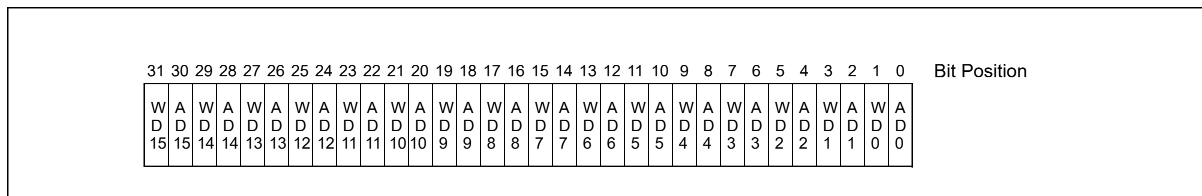


Figure 2-9. Protection Key Rights Register for User Pages (PKRU)

The layout of the PKRU register is shown in Figure 2-9. It contains 16 pairs of disable controls to prevent data accesses to user-mode linear addresses based on their protection keys. Each protection key *i* is associated with two bits in the PKRU register:

- Bit 2*i*, shown as "AD*i*" (access disable): if set, the processor prevents any data accesses to user-mode linear addresses with protection key *i*.
- Bit 2*i*+1, shown as "WD*i*" (write disable): if set, the processor prevents write accesses to user-mode linear addresses with protection key *i*.

See Section 4.6.2, "Protection Keys," for details of how the processor uses the PKRU register to control accesses to user-mode linear addresses.

2.8 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-3 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

Table 2-3. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	If CR4.UMIP = 1
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	If CR4.UMIP = 1
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	If CR4.UMIP = 1
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	If CR4.UMIP = 1
MOV CR*n*	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	If CR4.UMIP = 1
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes
ARPL	Adjust RPL	Yes[1, 5]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[3]	Read Time-Stamp Counter	Yes	Yes[2]
RDTSCP[7]	Read Serialized Time-Stamp Counter	Yes	Yes[2]

Table 2-3. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?
XGETBV	Return the state of XCR0	Yes	No
XSETBV	Enable one or more processor extended states	No[6]	Yes

NOTES:

1. Useful to application programs running at a CPL of 1 or 2.
2. The TSD and PCE flags in control register CR4 control access to these instructions by application programs running at a CPL of 3.
3. These instructions were introduced into the IA-32 Architecture with the Pentium processor.
4. This instruction was introduced into the IA-32 Architecture with the Pentium Pro processor and the Pentium processor with MMX technology.
5. This instruction is not supported in 64-bit mode.
6. Application uses XGETBV to query which set of processor extended states are enabled.
7. RDTSCP is introduced in Intel Core i7 processor.

2.8.1 Loading and Storing System Registers

The GDTR, LDTR, IDTR, and TR registers each have a load and store instruction for loading data into and storing data from the register:

- **LGDT (Load GDTR Register)** — Loads the GDT base address and limit from memory into the GDTR register.
- **SGDT (Store GDTR Register)** — Stores the GDT base address and limit from the GDTR register into memory.
- **LIDT (Load IDTR Register)** — Loads the IDT base address and limit from memory into the IDTR register.
- **SIDT (Store IDTR Register)** — Stores the IDT base address and limit from the IDTR register into memory.
- **LLDT (Load LDTR Register)** — Loads the LDT segment selector and segment descriptor from memory into the LDTR. (The segment selector operand can also be located in a general-purpose register.)
- **SLDT (Store LDTR Register)** — Stores the LDT segment selector from the LDTR register into memory or a general-purpose register.
- **LTR (Load Task Register)** — Loads segment selector and segment descriptor for a TSS from memory into the task register. (The segment selector operand can also be located in a general-purpose register.)
- **STR (Store Task Register)** — Stores the segment selector for the current task TSS from the task register into memory or a general-purpose register.

The LMSW (load machine status word) and SMSW (store machine status word) instructions operate on bits 0 through 15 of control register CR0. These instructions are provided for compatibility with the 16-bit Intel 286 processor. Programs written to run on 32-bit IA-32 processors should not use these instructions. Instead, they should access the control register CR0 using the MOV CR instruction.

The CLTS (clear TS flag in CR0) instruction is provided for use in handling a device-not-available exception (#NM) that occurs when the processor attempts to execute a floating-point instruction when the TS flag is set. This instruction allows the TS flag to be cleared after the x87 FPU context has been saved, preventing further #NM exceptions. See Section 2.5, "Control Registers," for more information on the TS flag.

The control registers (CR0, CR1, CR2, CR3, CR4, and CR8) are loaded using the MOV instruction. The instruction loads a control register from a general-purpose register or stores the content of a control register in a general-purpose register.

2.8.2 Verifying of Access Privileges

The processor provides several instructions for examining segment selectors and segment descriptors to determine if access to their associated segments is allowed. These instructions duplicate some of the automatic access rights and type checking done by the processor, thus allowing operating-system or executive software to prevent exceptions from being generated.

The ARPL (adjust RPL) instruction adjusts the RPL (requestor privilege level) of a segment selector to match that of the program or procedure that supplied the segment selector. See Section 5.10.4, "Checking Caller Access Privileges (ARPL Instruction)" for a detailed explanation of the function and use of this instruction. Note that ARPL is not supported in 64-bit mode.

The LAR (load access rights) instruction verifies the accessibility of a specified segment and loads access rights information from the segment's segment descriptor into a general-purpose register. Software can then examine the access rights to determine if the segment type is compatible with its intended use. See Section 5.10.1, "Checking Access Rights (LAR Instruction)" for a detailed explanation of the function and use of this instruction.

The LSL (load segment limit) instruction verifies the accessibility of a specified segment and loads the segment limit from the segment's segment descriptor into a general-purpose register. Software can then compare the segment limit with an offset into the segment to determine whether the offset lies within the segment. See Section 5.10.3, "Checking That the Pointer Offset Is Within Limits (LSL Instruction)" for a detailed explanation of the function and use of this instruction.

The VERR (verify for reading) and VERW (verify for writing) instructions verify if a selected segment is readable or writable, respectively, at a given CPL. See Section 5.10.2, "Checking Read/Write Rights (VERR and VERW Instructions)" for a detailed explanation of the function and use of these instructions.

2.8.3 Loading and Storing Debug Registers

Internal debugging facilities in the processor are controlled by a set of 8 debug registers (DR0-DR7). The MOV instruction allows setup data to be loaded to and stored from these registers.

On processors that support Intel 64 architecture, debug registers DR0-DR7 are 64 bits. In 32-bit modes and compatibility mode, writes to a debug register fill the upper 32 bits with zeros. Reads return the lower 32 bits. In 64-bit mode, the upper 32 bits of DR6-DR7 are reserved and must be written with zeros. Writing one to any of the upper 32 bits causes an exception, #GP(0).

In 64-bit mode, MOV DRn instructions read or write all 64 bits of a debug register (operand-size prefixes are ignored). All 64 bits of DR0-DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0-DR3 are in the limits of the implementation. Address matching is supported only on valid addresses generated by the processor implementation.

2.8.4 Invalidating Caches and TLBs

The processor provides several instructions for use in explicitly invalidating its caches and TLB entries. The INVD (invalidate cache with no writeback) instruction invalidates all data and instruction entries in the internal caches and sends a signal to the external caches indicating that they should also be invalidated.

The WBINVD (invalidate cache with writeback) instruction performs the same function as the INVD instruction, except that it writes back modified lines in its internal caches to memory before it invalidates the caches. After invalidating the caches local to the executing logical processor or processor core, WBINVD signals caches higher in the cache hierarchy (caches shared with the invalidating logical processor or core) to write back any data they have in modified state at the time of instruction execution and to invalidate their contents.

Note, non-shared caches may not be written back nor invalidated. In Figure 2-10 below, if code executing on either LP0 or LP1 were to execute a WBINVD, the shared L1 and L2 for LP0/LP1 will be written back and invalidated as will the shared L3. However, the L1 and L2 caches not shared with LP0 and LP1 will not be written back nor invalidated.

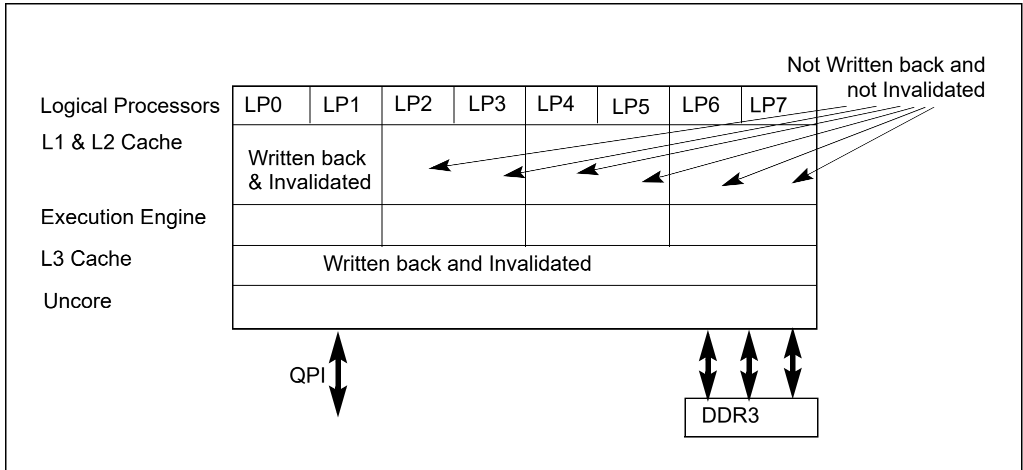


Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy

The INVLPG (invalidate TLB entry) instruction invalidates (flushes) the TLB entry for a specified page.

2.8.5 Controlling the Processor

The HLT (halt processor) instruction stops the processor until an enabled interrupt (such as NMI or SMI, which are normally enabled), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal is received. The processor generates a special bus cycle to indicate that the halt mode has been entered.

Hardware may respond to this signal in a number of ways. An indicator light on the front panel may be turned on. An NMI interrupt for recording diagnostic information may be generated. Reset initialization may be invoked (note that the BINIT# pin was introduced with the Pentium Pro processor). If any non-wake events are pending during shutdown, they will be handled after the wake event from shutdown is processed (for example, A20M# interrupts).

The LOCK prefix invokes a locked (atomic) read-modify-write operation when modifying a memory operand. This mechanism is used to allow reliable communications between processors in multiprocessor systems, as described below:

- In the Pentium processor and earlier IA-32 processors, the LOCK prefix causes the processor to assert the LOCK# signal during the instruction. This always causes an explicit bus lock to occur.
- In the Pentium 4, Intel Xeon, and P6 family processors, the locking operation is handled with either a cache lock or bus lock. If a memory access is cacheable and affects only a single cache line, a cache lock is invoked and the system bus and the actual memory location in system memory are not locked during the operation. Here, other Pentium 4, Intel Xeon, or P6 family processors on the bus write-back any modified data and invalidate their caches as necessary to maintain system memory coherency. If the memory access is not cacheable and/or it crosses a cache line boundary, the processor's LOCK# signal is asserted and the processor does not respond to requests for bus control during the locked operation.

The RSM (return from SMM) instruction restores the processor (from a context dump) to the state it was in prior to a system management mode (SMM) interrupt.

2.8.6 Reading Performance-Monitoring and Time-Stamp Counters

The RDPMC (read performance-monitoring counter) and RDTSC (read time-stamp counter) instructions allow application programs to read the processor's performance-monitoring and time-stamp counters, respectively. Processors based on Intel NetBurst® microarchitecture have eighteen 40-bit performance-monitoring counters; P6 family processors have two 40-bit counters. Intel® Atom™ processors and most of the processors based on the Intel Core microarchitecture support two types of performance monitoring counters: programmable performance counters similar to those available in the P6 family, and three fixed-function performance monitoring counters.

Details of programmable and fixed-function performance monitoring counters for each processor generation are described in Chapter 18, “Performance Monitoring”.

The programmable performance counters can support counting either the occurrence or duration of events. Events that can be monitored on programmable counters generally are model specific (except for architectural performance events enumerated by CPUID leaf 0AH); they may include the number of instructions decoded, interrupts received, or the number of cache loads. Individual counters can be set up to monitor different events. Use the system instruction WRMSR to set up values in one of the IA32_PERFEVTSELx MSR, in one of the 45 ESCRs and one of the 18 CCCR MSRs (for Pentium 4 and Intel Xeon processors); or in the PerfEvtSel0 or the PerfEvtSel1 MSR (for the P6 family processors). The RDPMC instruction loads the current count from the selected counter into the EDX:EAX registers.

Fixed-function performance counters record only specific events that are defined in Chapter 19, “Performance Monitoring Events”, and the width/number of fixed-function counters are enumerated by CPUID leaf 0AH.

The time-stamp counter is a model-specific 64-bit counter that is reset to zero each time the processor is reset. If not reset, the counter will increment ~9.5 x 10^{16} times per year when the processor is operating at a clock rate of 3GHz. At this clock frequency, it would take over 190 years for the counter to wrap around. The RDTSC instruction loads the current count of the time-stamp counter into the EDX:EAX registers.

See Section 18.1, “Performance Monitoring Overview,” and Section 17.16, “Time-Stamp Counter,” for more information about the performance monitoring and time-stamp counters.

The RDTSC instruction was introduced into the IA-32 architecture with the Pentium processor. The RDPMC instruction was introduced into the IA-32 architecture with the Pentium Pro processor and the Pentium processor with MMX technology. Earlier Pentium processors have two performance-monitoring counters, but they can be read only with the RDMSR instruction, and only at privilege level 0.

2.8.6.1 Reading Counters in 64-Bit Mode

In 64-bit mode, RDTSC operates the same as in protected mode. The count in the time-stamp counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

RDPMC requires an index to specify the offset of the performance-monitoring counter. In 64-bit mode for Pentium 4 or Intel Xeon processor families, the index is specified in ECX[30:0]. The current count of the performance-monitoring counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

2.8.7 Reading and Writing Model-Specific Registers

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions allow a processor’s 64-bit model-specific registers (MSRs) to be read and written, respectively. The MSR to be read or written is specified by the value in the ECX register.

RDMSR reads the value from the specified MSR to the EDX:EAX registers; WRMSR writes the value in the EDX:EAX registers to the specified MSR. RDMSR and WRMSR were introduced into the IA-32 architecture with the Pentium processor.

See Section 9.4, “Model-Specific Registers (MSRs),” for more information.

2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode

RDMSR and WRMSR require an index to specify the address of an MSR. In 64-bit mode, the index is 32 bits; it is specified using ECX.

2.8.8 Enabling Processor Extended States

The XSETBV instruction is required to enable OS support of individual processor extended states in XCR0 (see Section 2.6).

9. Updates to Chapter 4, Volume 3A

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Changes to this chapter: Fixed typo in section 4.6.2 "Protection Keys". Updated term "IA-32e paging" to "4-level paging"; included footnote on first usage of new term.

CHAPTER 4
PAGING

Chapter 3 explains how segmentation converts logical addresses to linear addresses. **Paging** (or linear-address translation) is the process of translating linear addresses so that they can be used to access memory or I/O devices. Paging translates each linear address to a **physical address** and determines, for each translation, what accesses to the linear address are allowed (the address's **access rights**) and the type of caching used for such accesses (the address's **memory type**).

Intel-64 processors support three different paging modes. These modes are identified and defined in Section 4.1. Section 4.2 gives an overview of the translation mechanism that is used in all modes. Section 4.3, Section 4.4, and Section 4.5 discuss the three paging modes in detail.

Section 4.6 details how paging determines and uses access rights. Section 4.7 discusses exceptions that may be generated by paging (page-fault exceptions). Section 4.8 considers data which the processor writes in response to linear-address accesses (accessed and dirty flags).

Section 4.9 describes how paging determines the memory types used for accesses to linear addresses. Section 4.10 provides details of how a processor may cache information about linear-address translation. Section 4.11 outlines interactions between paging and certain VMX features. Section 4.12 gives an overview of how paging can be used to implement virtual memory.

4.1 PAGING MODES AND CONTROL BITS

Paging behavior is controlled by the following control bits:

- The WP and PG flags in control register CR0 (bit 16 and bit 31, respectively).
- The PSE, PAE, PGE, PCIDE, SMEP, SMAP, and PKE flags in control register CR4 (bit 4, bit 5, bit 7, bit 17, bit 20, bit 21, and bit 22, respectively).
- The LME and NXE flags in the IA32_EFER MSR (bit 8 and bit 11, respectively).
- The AC flag in the EFLAGS register (bit 18).

Software enables paging by using the MOV to CR0 instruction to set CR0.PG. Before doing so, software should ensure that control register CR3 contains the physical address of the first paging structure that the processor will use for linear-address translation (see Section 4.2) and that structure is initialized as desired. See Table 4-3, Table 4-7, and Table 4-12 for the use of CR3 in the different paging modes.

Section 4.1.1 describes how the values of CR0.PG, CR4.PAE, and IA32_EFER.LME determine whether paging is in use and, if so, which of three paging modes is in use. Section 4.1.2 explains how to manage these bits to establish or make changes in paging modes. Section 4.1.3 discusses how CR0.WP, CR4.PSE, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, and IA32_EFER.NXE modify the operation of the different paging modes.

4.1.1 Three Paging Modes

If CR0.PG = 0, paging is not used. The logical processor treats all linear addresses as if they were physical addresses. CR4.PAE and IA32_EFER.LME are ignored by the processor, as are CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, CR4.SMAP, and IA32_EFER.NXE.

Paging is enabled if CR0.PG = 1. Paging can be enabled only if protection is enabled (CR0.PE = 1). If paging is enabled, one of three paging modes is used. The values of CR4.PAE and IA32_EFER.LME determine which paging mode is used:

- If CR0.PG = 1 and CR4.PAE = 0, **32-bit paging** is used. 32-bit paging is detailed in Section 4.3. 32-bit paging uses CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, and CR4.SMAP as described in Section 4.1.3.
- If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0, **PAE paging** is used. PAE paging is detailed in Section 4.4. PAE paging uses CR0.WP, CR4.PGE, CR4.SMEP, CR4.SMAP, and IA32_EFER.NXE as described in Section 4.1.3.

- If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1, **4-level paging**[1] is used.[2] 4-level paging is detailed in Section 4.5. 4-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, and IA32_EFER.NXE as described in Section 4.1.3. 4-level paging is available only on processors that support the Intel 64 architecture.

The three paging modes differ with regard to the following details:

- Linear-address width. The size of the linear addresses that can be translated.
- Physical-address width. The size of the physical addresses produced by paging.
- Page size. The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.
- Support for execute-disable access rights. In some paging modes, software can be prevented from fetching instructions from pages that are otherwise readable.
- Support for PCIDs. With 4-level paging, software can enable a facility by which a logical processor caches information for multiple linear-address spaces. The processor may retain cached information when software switches between different linear-address spaces.
- Support for protection keys. With 4-level paging, software can enable a facility by which each linear address is associated with a **protection key**. Software can use a new control register to determine, for each protection keys, how software can access linear addresses associated with that protection key.

Table 4-1 illustrates the principal differences between the three paging modes.

Table 4-1. Properties of Different Paging Modes

Paging Mode	PG in CR0	PAE in CR4	LME in IA32_EFER	Lin.-Addr. Width	Phys.-Addr. Width[1]	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
None	0	N/A	N/A	32	32	N/A	No	No
32-bit	1	0	0[2]	32	Up to 40[3]	4 KB 4 MB[4]	No	No
PAE	1	1	0	32	Up to 52	4 KB 2 MB	Yes[5]	No
4-level	1	1	1	48	Up to 52	4 KB 2 MB 1 GB[6]	Yes[5]	Yes[7]

NOTES:

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support 4-level paging support 1-GByte pages; see Section 4.1.4.
7. PCIDs are used only if CR4.PCIDE = 1; see Section 4.10.1. Protection keys are used only if certain conditions hold; see Section 4.6.2.

---

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.
2. The LMA flag in the IA32_EFER MSR (bit 10) is a status bit that indicates whether the logical processor is in IA-32e mode (and thus using 4-level paging). The processor always sets IA32_EFER.LMA to CR0.PG & IA32_EFER.LME. Software cannot directly modify IA32_EFER.LMA; an execution of WRMSR to the IA32_EFER MSR ignores bit 10 of its source operand.

Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging are used only in legacy protected mode. Because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

Because it is used only if IA32_EFER.LME = 1, 4-level paging is used only in IA-32e mode. (In fact, it is the use of 4-level paging that defines IA-32e mode.) IA-32e mode has two sub-modes:

- Compatibility mode. This mode uses only 32-bit linear addresses. 4-level paging treats bits 47:32 of such an address as all 0.
- 64-bit mode. While this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical.[1] 4-level paging does not use bits 63:48 of such addresses.

4.1.2 Paging-Mode Enabling

If CR0.PG = 1, a logical processor is in one of three paging modes, depending on the values of CR4.PAE and IA32_EFER.LME. Figure 4-1 illustrates how software can enable these modes and make transitions between them. The following items identify certain limitations and other details:

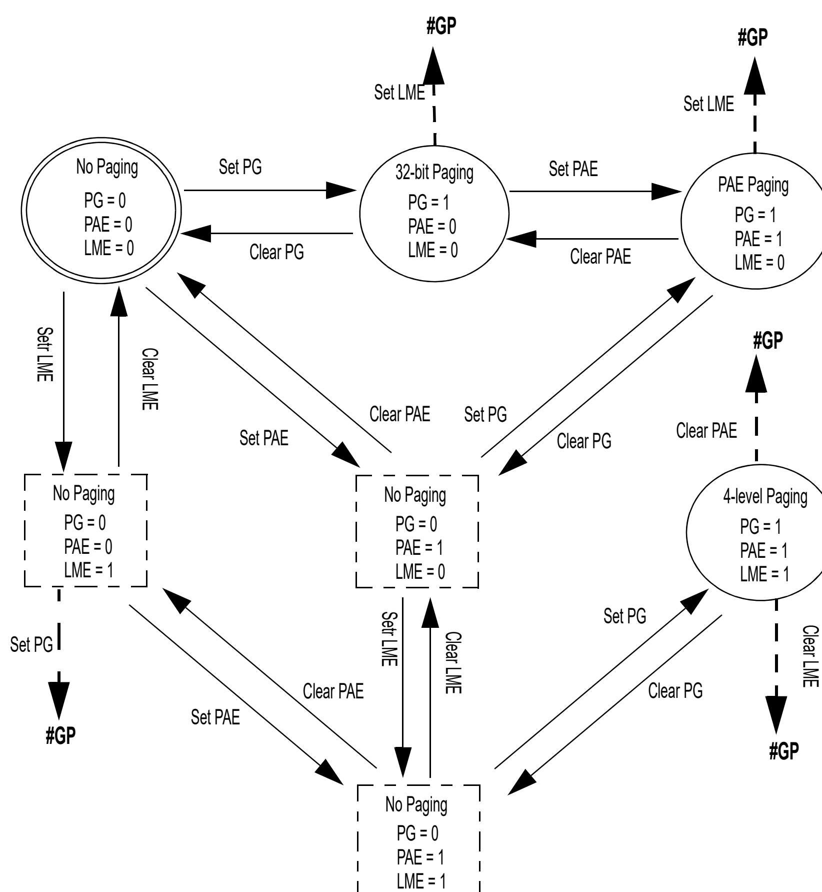


Figure 4-1. Enabling and Changing Paging Modes

- IA32_EFER.LME cannot be modified while paging is enabled (CR0.PG = 1). Attempts to do so using WRMSR cause a general-protection exception (#GP(0)).

1. Such an address is called **canonical**. Use of a non-canonical linear address in 64-bit mode produces a general-protection exception (#GP(0)); the processor does not attempt to translate non-canonical linear addresses using 4-level paging.

- Paging cannot be enabled (by setting CR0.PG to 1) while CR4.PAE = 0 and IA32_EFER.LME = 1. Attempts to do so using MOV to CR0 cause a general-protection exception (#GP(0)).
- CR4.PAE cannot be cleared while 4-level paging is active (CR0.PG = 1 and IA32_EFER.LME = 1). Attempts to do so using MOV to CR4 cause a general-protection exception (#GP(0)).
- Regardless of the current paging mode, software can disable paging by clearing CR0.PG with MOV to CR0.[1]
- Software can make transitions between 32-bit paging and PAE paging by changing the value of CR4.PAE with MOV to CR4.
- Software cannot make transitions directly between 4-level paging and either of the other two paging modes. It must first disable paging (by clearing CR0.PG with MOV to CR0), then set CR4.PAE and IA32_EFER.LME to the desired values (with MOV to CR4 and WRMSR), and then re-enable paging (by setting CR0.PG with MOV to CR0). As noted earlier, an attempt to clear either CR4.PAE or IA32_EFER.LME cause a general-protection exception (#GP(0)).
- VMX transitions allow transitions between paging modes that are not possible using MOV to CR or WRMSR. This is because VMX transitions can load CR0, CR4, and IA32_EFER in one operation. See Section 4.11.1.

4.1.3 Paging-Mode Modifiers

Details of how each paging mode operates are determined by the following control bits:

- The WP flag in CR0 (bit 16).
- The PSE, PGE, PCIDE, SMEP, SMAP, and PKE flags in CR4 (bit 4, bit 7, bit 17, bit 20, bit 21, and bit 22 respectively).
- The NXE flag in the IA32_EFER MSR (bit 11).

CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP = 0, supervisor-mode write accesses are allowed to linear addresses with read-only access rights; if CR0.WP = 1, they are not. (User-mode write accesses are never allowed to linear addresses with read-only access rights, regardless of the value of CR0.WP.) Section 4.6 explains how access rights are determined, including the definition of supervisor-mode and user-mode accesses.

CR4.PSE enables 4-MByte pages for 32-bit paging. If CR4.PSE = 0, 32-bit paging can use only 4-KByte pages; if CR4.PSE = 1, 32-bit paging can use both 4-KByte pages and 4-MByte pages. See Section 4.3 for more information. (PAE paging and 4-level paging can use multiple page sizes regardless of the value of CR4.PSE.)

CR4.PGE enables global pages. If CR4.PGE = 0, no translations are shared across address spaces; if CR4.PGE = 1, specified translations may be shared across address spaces. See Section 4.10.2.4 for more information.

CR4.PCIDE enables process-context identifiers (PCIDs) for 4-level paging (CR4.PCIDE can be 1 only when 4-level paging is in use). PCIDs allow a logical processor to cache information for multiple linear-address spaces. See Section 4.10.1 for more information.

CR4.SMEP allows pages to be protected from supervisor-mode instruction fetches. If CR4.SMEP = 1, software operating in supervisor mode cannot fetch instructions from linear addresses that are accessible in user mode. Section 4.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.SMAP allows pages to be protected from supervisor-mode data accesses. If CR4.SMAP = 1, software operating in supervisor mode cannot access data at linear addresses that are accessible in user mode. Software can override this protection by setting EFLAGS.AC. Section 4.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.PKE allows each linear address to be associated with a **protection key**. The PKRU register specifies, for each protection key, whether linear addresses with that protection key can be read or written by software. See Section 4.6 for more information.

IA32_EFER.NXE enables execute-disable access rights for PAE paging and 4-level paging. If IA32_EFER.NXE = 1, instruction fetches can be prevented from specified linear addresses (even if data reads from the addresses are

1. If CR4.PCIDE = 1, an attempt to clear CR0.PG causes a general-protection exception (#GP); software should clear CR4.PCIDE before attempting to disable paging.

allowed). Section 4.6 explains how access rights are determined. (IA32_EFER.NXE has no effect with 32-bit paging. Software that wants to use this feature to limit instruction fetches from readable pages must use either PAE paging or 4-level paging.)

4.1.4 Enumeration of Paging Features by CPUID

Software can discover support for different paging features using the CPUID instruction:

- PSE: page-size extensions for 32-bit paging.
  If CPUID.01H:EDX.PSE [bit 3] = 1, CR4.PSE may be set to 1, enabling support for 4-MByte pages with 32-bit paging (see Section 4.3).
- PAE: physical-address extension.
  If CPUID.01H:EDX.PAE [bit 6] = 1, CR4.PAE may be set to 1, enabling PAE paging (this setting is also required for 4-level paging).
- PGE: global-page support.
  If CPUID.01H:EDX.PGE [bit 13] = 1, CR4.PGE may be set to 1, enabling the global-page feature (see Section 4.10.2.4).
- PAT: page-attribute table.
  If CPUID.01H:EDX.PAT [bit 16] = 1, the 8-entry page-attribute table (PAT) is supported. When the PAT is supported, three bits in certain paging-structure entries select a memory type (used to determine type of caching used) from the PAT (see Section 4.9.2).
- PSE-36: page-size extensions with 40-bit physical-address extension.
  If CPUID.01H:EDX.PSE-36 [bit 17] = 1, the PSE-36 mechanism is supported, indicating that translations using 4-MByte pages with 32-bit paging may produce physical addresses with up to 40 bits (see Section 4.3).
- PCID: process-context identifiers.
  If CPUID.01H:ECX.PCID [bit 17] = 1, CR4.PCIDE may be set to 1, enabling process-context identifiers (see Section 4.10.1).
- SMEP: supervisor-mode execution prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMEP [bit 7] = 1, CR4.SMEP may be set to 1, enabling supervisor-mode execution prevention (see Section 4.6).
- SMAP: supervisor-mode access prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMAP [bit 20] = 1, CR4.SMAP may be set to 1, enabling supervisor-mode access prevention (see Section 4.6).
- PKU: protection keys.
  If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, CR4.PKE may be set to 1, enabling protection keys (see Section 4.6).
- NX: execute disable.
  If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing PAE paging and 4-level paging to disable execute access to selected pages (see Section 4.6). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.NXE to be set to 1.)
- Page1GB: 1-GByte pages.
  If CPUID.80000001H:EDX.Page1GB [bit 26] = 1, 1-GByte pages are supported with 4-level paging (see Section 4.5).
- LM: IA-32e mode support.
  If CPUID.80000001H:EDX.LM [bit 29] = 1, IA32_EFER.LME may be set to 1, enabling 4-level paging. (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.LME to be set to 1.)
- CPUID.80000008H:EAX[7:0] reports the physical-address width supported by the processor. (For processors that do not support CPUID function 80000008H, the width is generally 36 if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 otherwise.) This width is referred to as MAXPHYADDR. MAXPHYADDR is at most 52.
- CPUID.80000008H:EAX[15:8] reports the linear-address width supported by the processor. Generally, this value is 48 if CPUID.80000001H:EDX.LM [bit 29] = 1 and 32 otherwise. (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)

4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All three paging modes translate linear addresses using **hierarchical paging structures**. This section provides an overview of their operation. Section 4.3, Section 4.4, and Section 4.5 provide details for the three paging modes.

Every paging structure is 4096 Bytes in size and comprises a number of individual **entries**. With 32-bit paging, each entry is 32 bits (4 bytes); there are thus 1024 entries in each structure. With PAE paging and 4-level paging, each entry is 64 bits (8 bytes); there are thus 512 entries in each structure. (PAE paging includes one exception, a paging structure that is 32 bytes in size, containing 4 64-bit entries.)

The processor uses the upper portion of a linear address to identify a series of paging-structure entries. The last of these entries identifies the physical address of the region to which the linear address translates (called the **page frame**). The lower portion of the linear address (called the **page offset**) identifies the specific address within that region to which the linear address translates.

Each paging-structure entry contains a physical address, which is either the address of another paging structure or the address of a page frame. In the first case, the entry is said to **reference** the other paging structure; in the latter, the entry is said to **map a page**.

The first paging structure used for any translation is located at the physical address in CR3. A linear address is translated using the following iterative procedure. A portion of the linear address (initially the uppermost bits) selects an entry in a paging structure (initially the one located using CR3). If that entry references another paging structure, the process continues with that paging structure and with the portion of the linear address immediately below that just used. If instead the entry maps a page, the process completes: the physical address in the entry is that of the page frame and the remaining lower portion of the linear address is the page offset.

The following items give an example for each of the three paging modes (each example locates a 4-KByte page frame):

- With 32-bit paging, each paging structure comprises 1024 = 2^{10} entries. For this reason, the translation process uses 10 bits at a time from a 32-bit linear address. Bits 31:22 identify the first paging-structure entry and bits 21:12 identify a second. The latter identifies the page frame. Bits 11:0 of the linear address are the page offset within the 4-KByte page frame. (See Figure 4-2 for an illustration.)
- With PAE paging, the first paging structure comprises only 4 = 2^2 entries. Translation thus begins by using bits 31:30 from a 32-bit linear address to identify the first paging-structure entry. Other paging structures comprise 512 =2^9 entries, so the process continues by using 9 bits at a time. Bits 29:21 identify a second paging-structure entry and bits 20:12 identify a third. This last identifies the page frame. (See Figure 4-5 for an illustration.)
- With 4-level paging, each paging structure comprises 512 = 2^9 entries and translation uses 9 bits at a time from a 48-bit linear address. Bits 47:39 identify the first paging-structure entry, bits 38:30 identify a second, bits 29:21 a third, and bits 20:12 identify a fourth. Again, the last identifies the page frame. (See Figure 4-8 for an illustration.)

The translation process in each of the examples above completes by identifying a page frame; the page frame is part of the **translation** of the original linear address. In some cases, however, the paging structures may be configured so that the translation process terminates before identifying a page frame. This occurs if the process encounters a paging-structure entry that is marked “not present” (because its P flag — bit 0 — is clear) or in which a reserved bit is set. In this case, there is no translation for the linear address; an access to that address causes a page-fault exception (see Section 4.7).

In the examples above, a paging-structure entry maps a page with a 4-KByte page frame when only 12 bits remain in the linear address; entries identified earlier always reference other paging structures. That may not apply in other cases. The following items identify when an entry maps a page and when it references another paging structure:

- If more than 12 bits remain in the linear address, bit 7 (PS — page size) of the current paging-structure entry is consulted. If the bit is 0, the entry references another paging structure; if the bit is 1, the entry maps a page.
- If only 12 bits remain in the linear address, the current paging-structure entry always maps a page (bit 7 is used for other purposes).

If a paging-structure entry maps a page when more than 12 bits remain in the linear address, the entry identifies a page frame larger than 4 KBytes. For example, 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain. If that entry maps a page, the page frame is 2^{22} Bytes = 4 MBytes.

32-bit paging supports 4-MByte pages if CR4.PSE = 1. PAE paging and 4-level paging support 2-MByte pages (regardless of the value of CR4.PSE). 4-level paging may support 1-GByte pages (see Section 4.1.4).

Paging structures are given different names based on their uses in the translation process. Table 4-2 gives the names of the different paging structures. It also provides, for each structure, the source of the physical address used to locate it (CR3 or a different paging-structure entry); the bits in the linear address used to select an entry from the structure; and details of whether and how such an entry can map a page.

Table 4-2. Paging Structures in the Different Paging Modes

Paging Structure	Entry Name	Paging Mode	Physical Address of Structure	Bits Selecting Entry	Page Mapping
PML4 table	PML4E	32-bit, PAE	N/A		
		4-level	CR3	47:39	N/A (PS must be 0)
Page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A (PS must be 0)
		4-level	PML4E	38:30	1-GByte page if PS=1[1]
Page directory	PDE	32-bit	CR3	31:22	4-MByte page if PS=1[2]
		PAE, 4-level	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-KByte page
		PAE, 4-level		20:12	4-KByte page

NOTES:

1. Not all processors allow the PS flag to be 1 in PDPTEs; see Section 4.1.4 for how to determine whether 1-GByte pages are supported.
2. 32-bit paging ignores the PS flag in a PDE (and uses the entry to reference a page table) unless CR4.PSE = 1. Not all processors allow CR4.PSE to be 1; see Section 4.1.4 for how to determine whether 4-MByte pages are supported with 32-bit paging.

4.3 32-BIT PAGING

A logical processor uses 32-bit paging if CR0.PG = 1 and CR4.PAE = 0. 32-bit paging translates 32-bit linear addresses to 40-bit physical addresses.[1] Although 40 bits corresponds to 1 TByte, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

32-bit paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the page directory. Table 4-3 illustrates how CR3 is used with 32-bit paging.

32-bit paging may map linear addresses to either 4-KByte pages or 4-MByte pages. Figure 4-2 illustrates the translation process when it uses a 4-KByte page; Figure 4-3 covers the case of a 4-MByte page. The following items describe the 32-bit paging process in more detail as well has how the page size is determined:

- A 4-KByte naturally aligned page directory is located at the physical address specified in bits 31:12 of CR3 (see Table 4-3). A page directory comprises 1024 32-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from CR3.

1. Bits in the range 39:32 are 0 in any physical address used by 32-bit paging except those used to map 4-MByte pages. If the processor does not support the PSE-36 mechanism, this is true also for physical addresses used to map 4-MByte pages. If the processor does support the PSE-36 mechanism and MAXPHYADDR < 40, bits in the range 39:MAXPHYADDR are 0 in any physical address used to map a 4-MByte page. (The corresponding bits are reserved in PDEs.) See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

— Bits 11:2 are bits 31:22 of the linear address.

— Bits 1:0 are 0.

Because a PDE is identified using bits 31:22 of the linear address, it controls access to a 4-Mbyte region of the linear-address space. Use of the PDE depends on CR4.PSE and the PDE's PS flag (bit 7):

- If CR4.PSE = 1 and the PDE's PS flag is 1, the PDE maps a 4-MByte page (see Table 4-4). The final physical address is computed as follows:
  - Bits 39:32 are bits 20:13 of the PDE.
  - Bits 31:22 are bits 31:22 of the PDE.[1]
  - Bits 21:0 are from the original linear address.
- If CR4.PSE = 0 or the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 31:12 of the PDE (see Table 4-5). A page table comprises 1024 32-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PDE.
  - Bits 11:2 are bits 21:12 of the linear address.
  - Bits 1:0 are 0.
- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-6). The final physical address is computed as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

With 32-bit paging, there are reserved bits only if CR4.PSE = 1:

- If the P flag and the PS flag (bit 7) of a PDE are both 1, the bits reserved depend on MAXPHYADDR, and whether the PSE-36 mechanism is supported:[2]
  - If the PSE-36 mechanism is not supported, bits 21:13 are reserved.
  - If the PSE-36 mechanism is supported, bits 21:(M–19) are reserved, where M is the minimum of 40 and MAXPHYADDR.
- If the PAT is not supported:[3]
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

(If CR4.PSE = 0, no bits are reserved with 32-bit paging.)

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

---

1. The upper bits in the final physical address do not all come from corresponding positions in the PDE; the physical-address bits in the PDE are not all contiguous.
2. See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.
3. See Section 4.1.4 for how to determine whether the PAT is supported.

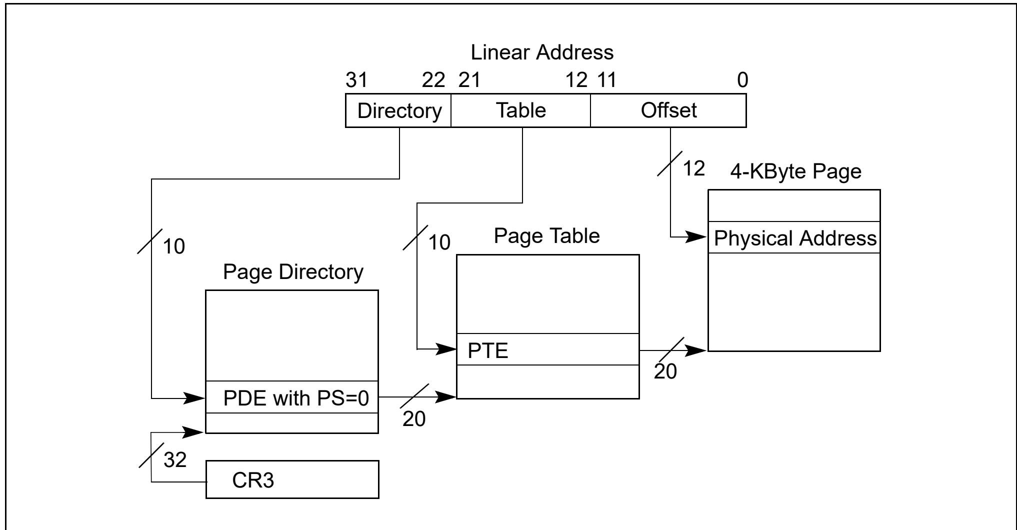


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

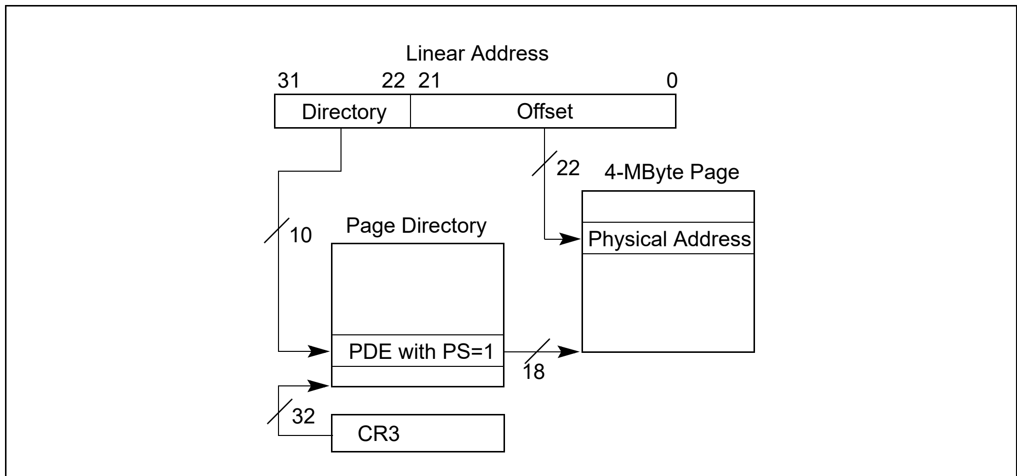


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="20">Address of page directory[1]</td><td colspan="7">Ignored</td><td>PCD</td><td>PWT</td><td colspan="3">Ignored</td><td>CR3</td></tr>
<tr><td colspan="10">Bits 31:22 of address of 4MB page frame</td><td colspan="5">Reserved (must be 0)</td><td colspan="4">Bits 39:32 of address[2]</td><td>PAT</td><td colspan="3">Ignored</td><td>G</td><td>1</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: 4MB page</td></tr>
<tr><td colspan="20">Address of page table</td><td colspan="4">Ignored</td><td>0</td><td>Ign</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: page table</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PDE: not present</td></tr>
<tr><td colspan="20">Address of 4KB page frame</td><td colspan="3">Ignored</td><td>G</td><td>PAT</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PTE: 4KB page</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PTE: not present</td></tr>
</table>

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

Table 4-3. Use of CR3 with 32-Bit Paging

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
11:5	Ignored
31:12	Physical address of the 4-KByte aligned page directory used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

Table 4-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 4-5)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
12 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
(M–20):13	Bits (M–1):32 of physical address of the 4-MByte page referenced by this entry[2]
21:(M–19)	Reserved (must be 0)
31:22	Bits 31:22 of physical address of the 4-MByte page referenced by this entry

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.
2. If the PSE-36 mechanism is not supported, M is 32, and this row does not apply. If the PSE-36 mechanism is supported, M is the minimum of 40 and MAXPHYADDR (this row does not apply if MAXPHYADDR = 32). See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.

4.4 PAE PAGING

A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0. PAE paging translates 32-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

With PAE paging, a logical processor maintains a set of four (4) PDPTE registers, which are loaded from an address in CR3. Linear address are translated using 4 hierarchies of in-memory paging structures, each located using one of the PDPTE registers. (This is different from the other paging modes, in which there is one hierarchy referenced by CR3.)

Section 4.4.1 discusses the PDPTE registers. Section 4.4.2 describes linear-address translation with PAE paging.

4.4.1 PDPTE Registers

When PAE paging is used, CR3 references the base of a 32-Byte **page-directory-pointer table**. Table 4-7 illustrates how CR3 is used with PAE paging.

Table 4-7. Use of CR3 with PAE Paging

Bit Position(s)	Contents
4:0	Ignored
31:5	Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

The page-directory-pointer-table comprises four (4) 64-bit entries called PDPTEs. Each PDPTE controls access to a 1-GByte region of the linear-address space. Corresponding to the PDPTEs, the logical processor maintains a set of four (4) internal, non-architectural PDPTE registers, called PDPTE0, PDPTE1, PDPTE2, and PDPTE3. The logical processor loads these registers from the PDPTEs in memory as part of certain operations:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, CR4.PSE, or CR4.SMEP; then the PDPTEs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTEs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTEs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPTE registers. See Section 4.11.1.

Table 4-8 gives the format of a PDPTE. If any of the PDPTEs sets both the P flag (bit 0) and any reserved bit, the MOV to CR instruction causes a general-protection exception (#GP(0)) and the PDPTEs are not loaded.[2] As shown in Table 4-8, bits 2:1, 8:5, and 63:MAXPHYADDR are reserved in the PDPTEs.

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by PAE paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.
2. On some processors, reserved bits are checked even in PDPTEs in which the P flag (bit 0) is 0.

Table 4-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

NOTES:

1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

4.4.2 Linear-Address Translation with PAE Paging

PAE paging may map linear addresses to either 4-KByte pages or 2-MByte pages. Figure 4-5 illustrates the translation process when it produces a 4-KByte page; Figure 4-6 covers the case of a 2-MByte page. The following items describe the PAE paging process in more detail as well has how the page size is determined:

- Bits 31:30 of the linear address select a PDPTE register (see Section 4.4.1); this is PDPTE*i*, where *i* is the value of bits 31:30.[1] Because a PDPTE register is identified using bits 31:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. If the P flag (bit 0) of PDPTE*i* is 0, the processor ignores bits 63:1, and there is no mapping for the 1-GByte region controlled by PDPTE*i*. A reference using a linear address in this region causes a page-fault exception (see Section 4.7).
- If the P flag of PDPTE*i* is 1, 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of PDPTE*i* (see Table 4-8 in Section 4.4.1). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 51:12 are from PDPTE*i*.
  - Bits 11:3 are bits 29:21 of the linear address.
  - Bits 2:0 are 0.

Because a PDE is identified using bits 31:21 of the linear address, it controls access to a 2-Mbyte region of the linear-address space. Use of the PDE depends on its PS flag (bit 7):

- If the PDE's PS flag is 1, the PDE maps a 2-MByte page (see Table 4-9). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.
- If the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 4-10). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.

1. With PAE paging, the processor does not use CR3 when translating a linear address (as it does in the other paging modes). It does not access the PDPTEs in the page-directory-pointer table during linear-address translation.

— Bits 11:3 are bits 20:12 of the linear address.

— Bits 2:0 are 0.

- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-11). The final physical address is computed as follows:

  — Bits 51:12 are from the PTE.

  — Bits 11:0 are from the original linear address.

If the P flag (bit 0) of a PDE or a PTE is 0 or if a PDE or a PTE sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

The following bits are reserved with PAE paging:

- If the P flag (bit 0) of a PDE or a PTE is 1, bits 62:MAXPHYADDR are reserved.
- If the P flag and the PS flag (bit 7) of a PDE are both 1, bits 20:13 are reserved.
- If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.
- If the PAT is not supported:[1]

  — If the P flag of a PTE is 1, bit 7 is reserved.

  — If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

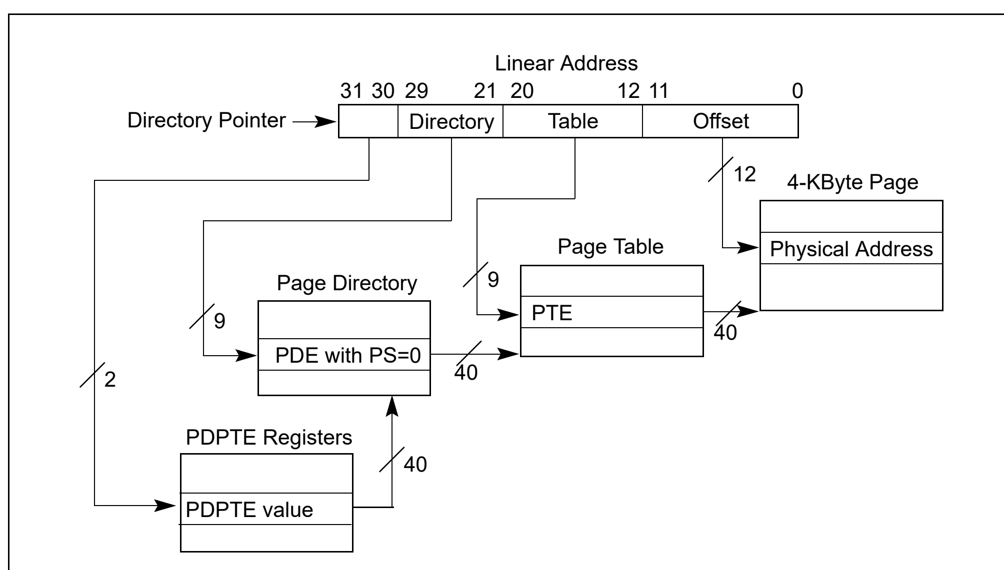


Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

1. See Section 4.1.4 for how to determine whether the PAT is supported.

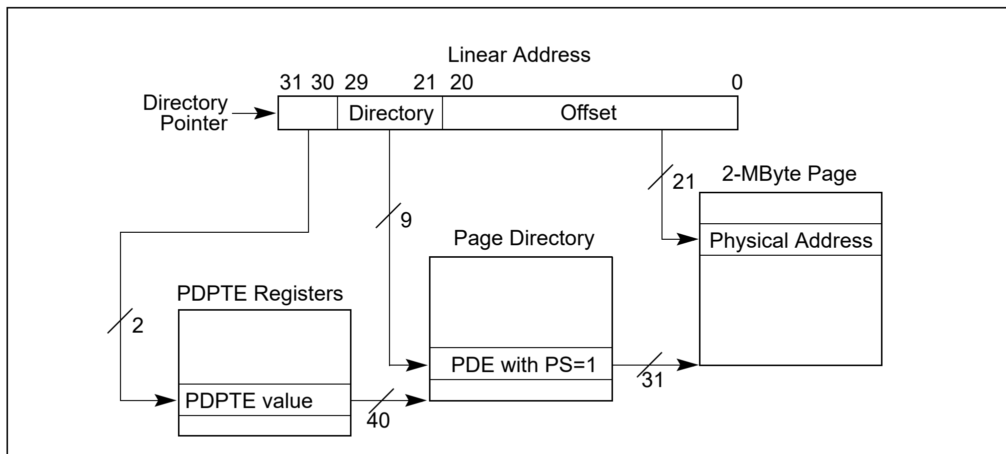


Figure 4-6. Linear-Address Translation to a 2-MByte Page using PAE Paging

Table 4-9. Format of a PAE Page-Directory Entry that Maps a 2-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 2-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 4-10)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
12 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
20:13	Reserved (must be 0)
(M–1):21	Physical address of the 2-MByte page referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.

Table 4-10. Format of a PAE Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-9)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page table referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page (Contd.)

Bit Position(s)	Contents
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.

Figure 4-7 gives a summary of the formats of CR3 and the paging-structure entries with PAE paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

Bit positions: 63 62 61 60 59 58 57 56 55 54 53 52 51 … M[1] M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Fields (from bit 63 down to bit 0)	Entry
Ignored[2] | Address of page-directory-pointer table | Ignored	CR3
Reserved[3] | Address of page directory | Ign. | Rsvd. | PCD | PWT | Rsvd | 1	PDPTE: present
Ignored | 0	PDTPE: not present
XD[4] | Reserved | Address of 2MB page frame | Reserved | PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1	PDE: 2MB page
XD | Reserved | Address of page table | Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1	PDE: page table
Ignored | 0	PDE: not present
XD | Reserved | Address of 4KB page frame | Ign. | G | PAT | D | A | PCD | PWT | U/S | R/W | 1	PTE: 4KB page
Ignored | 0	PTE: not present

Figure 4-7. Formats of CR3 and Paging-Structure Entries with PAE Paging

NOTES:

1. M is an abbreviation for MAXPHYADDR.
2. CR3 has 64 bits only on processors supporting the Intel-64 architecture. These bits are ignored with PAE paging.
3. Reserved fields must be 0.
4. If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.

4.5 4-LEVEL PAGING

A logical processor uses 4-level paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With 4-level paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. 4-level paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

4-level paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with 4-level paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

- Table 4-12 illustrates how CR3 is used with 4-level paging if CR4.PCIDE = 0.

Table 4-12. Use of CR3 with 4-Level Paging and CR4.PCIDE = 0

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
11:5	Ignored
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[1]
63:M	Reserved (must be 0)

NOTES:

1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

- Table 4-13 illustrates how CR3 is used with 4-level paging if CR4.PCIDE = 1.

Table 4-13. Use of CR3 with 4-Level Paging and CR4.PCIDE = 1

Bit Position(s)	Contents
11:0	PCID (see Section 4.10.1)[1]
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[2]
63:M	Reserved (must be 0)[3]

NOTES:

1. Section 4.9.2 explains how the processor determines the memory type used to access the PML4 table during linear-address translation with CR4.PCIDE = 1.
2. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.
3. See Section 4.10.4.1 for use of bit 63 of the source operand of the MOV to CR3 instruction.

After software modifies the value of CR4.PCIDE, the logical processor immediately begins using CR3 as specified for the new value. For example, if software changes CR4.PCIDE from 1 to 0, the current PCID immediately changes

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by 4-level paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.

from CR3[11:0] to 000H (see also Section 4.10.4.1). In addition, the logical processor subsequently determines the memory type used to access the PML4 table using CR3.PWT and CR3.PCD, which had been bits 4:3 of the PCID.

4-level paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.[1] Figure 4-8 illustrates the translation process when it produces a 4-KByte page; Figure 4-9 covers the case of a 2-MByte page, and Figure 4-10 the case of a 1-GByte page.

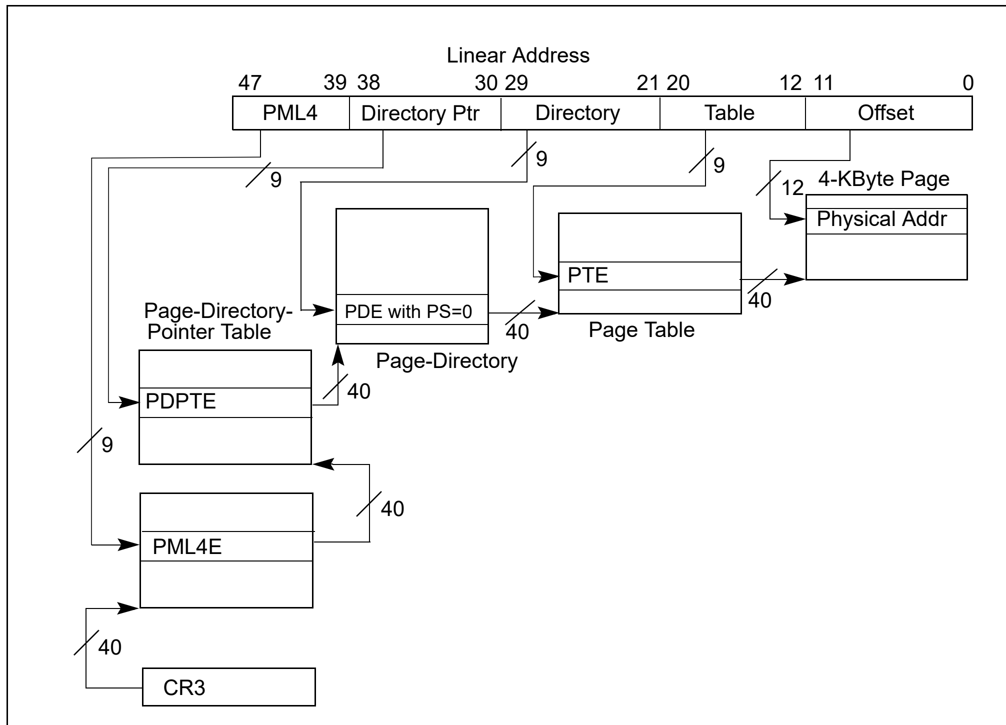


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

1. Not all processors support 1-GByte pages; see Section 4.1.4.

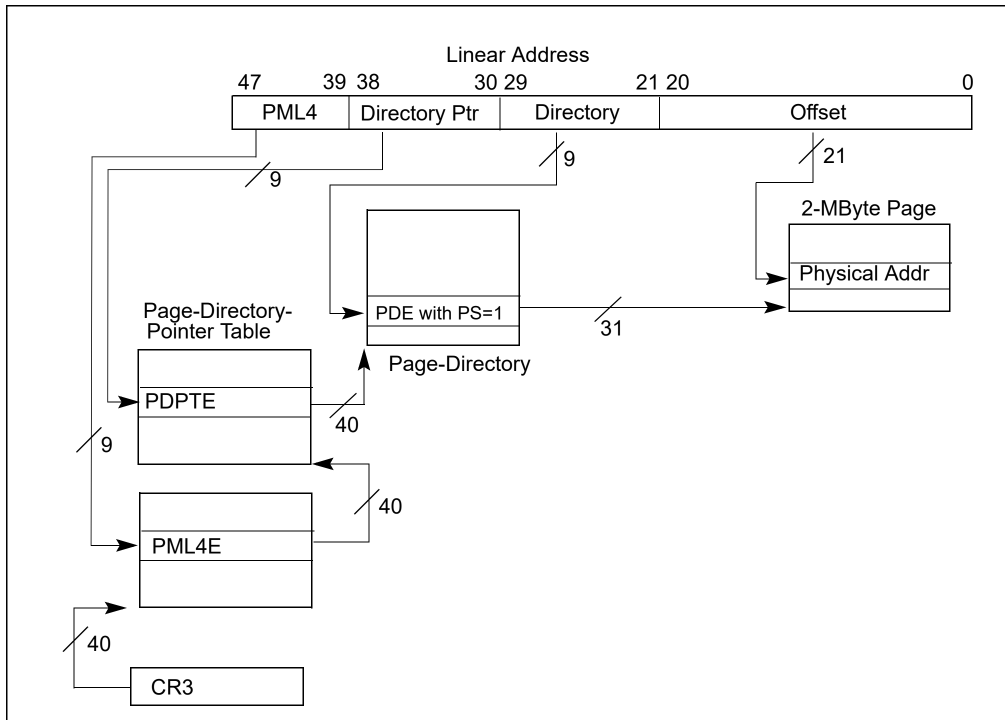


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

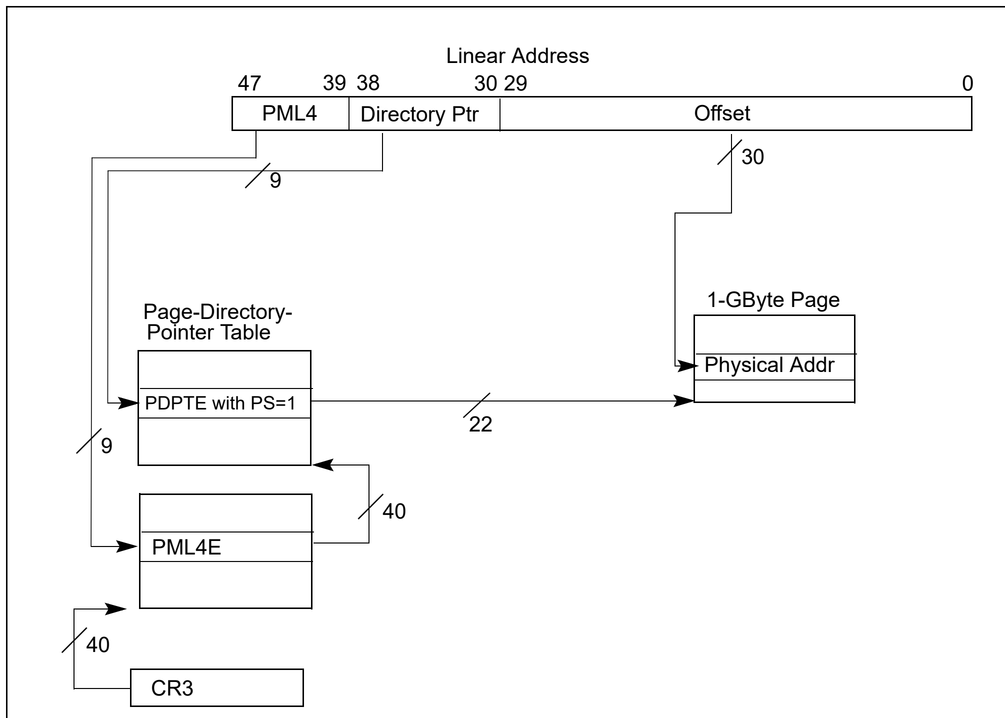


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

If CR4.PKE = 1, 4-level paging associates with each linear address a **protection key**. Section 4.6 explains how the processor uses the protection key in its determination of the access rights of each linear address.

The following items describe the 4-level paging process in more detail as well has how the page size and protection key are determined.

- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.
- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-14). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.
  - Bits 11:3 are bits 38:30 of the linear address.
  - Bits 2:0 are all 0.

Because a PDPTE is identified using bits 47:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. Use of the PDPTE depends on its PS flag (bit 7):[1]

- If the PDPTE's PS flag is 1, the PDPTE maps a 1-GByte page (see Table 4-15). The final physical address is computed as follows:
  - Bits 51:30 are from the PDPTE.
  - Bits 29:0 are from the original linear address.

  If CR4.PKE = 1, the linear address's protection key is the value of bits 62:59 of the PDPTE.
- If the PDPTE's PS flag is 0, a 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of the PDPTE (see Table 4-16). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDPTE.
  - Bits 11:3 are bits 29:21 of the linear address.
  - Bits 2:0 are all 0.

Because a PDE is identified using bits 47:21 of the linear address, it controls access to a 2-MByte region of the linear-address space. Use of the PDE depends on its PS flag:

- If the PDE's PS flag is 1, the PDE maps a 2-MByte page (see Table 4-17). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.

  If CR4.PKE = 1, the linear address's protection key is the value of bits 62:59 of the PDE.
- If the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 4-18). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.
  - Bits 11:3 are bits 20:12 of the linear address.
  - Bits 2:0 are all 0.

1. The PS flag of a PDPTE is reserved and must be 0 (if the P flag is 1) if 1-GByte pages are not supported. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.

- Because a PTE is identified using bits 47:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-19). The final physical address is computed as follows:
  - Bits 51:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

  If CR4.PKE = 1, the linear address's protection key is the value of bits 62:59 of the PTE.

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

The following bits are reserved with 4-level paging:

- If the P flag of a paging-structure entry is 1, bits 51:MAXPHYADDR are reserved.
- If the P flag of a PML4E is 1, the PS flag is reserved.
- If 1-GByte pages are not supported and the P flag of a PDPTE is 1, the PS flag is reserved.[1]
- If the P flag and the PS flag of a PDPTE are both 1, bits 29:13 are reserved.
- If the P flag and the PS flag of a PDE are both 1, bits 20:13 are reserved.
- If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

Figure 4-11 gives a summary of the formats of CR3 and the 4-level paging-structure entries. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

Table 4-14. Format of a 4-Level PML4 Entry (PML4E) that References a Page-Directory-Pointer Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page-directory-pointer table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Reserved (must be 0)
11:8	Ignored

1. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.

Table 4-14. Format of a 4-Level PML4 Entry (PML4E) that References a Page-Directory-Pointer Table (Contd.)

Bit Position(s)	Contents
M–1:12	Physical address of 4-KByte aligned page-directory-pointer table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 512-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-15. Format of a 4-Level Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 1-GByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page directory; see Table 4-16)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
12 (PAT)	Indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)[1]
29:13	Reserved (must be 0)
(M–1):30	Physical address of the 1-GByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

NOTES:

1. The PAT is supported on all processors that support 4-level paging.

Table 4-16. Format of a 4-Level Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 4-15)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-17. Format of a 4-Level Page-Directory Entry that Maps a 2-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 2-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 4-18)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Table 4-17. Format of a 4-Level Page-Directory Entry that Maps a 2-MByte Page (Contd.)

Bit Position(s)	Contents
11:9	Ignored
12 (PAT)	Indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
20:13	Reserved (must be 0)
(M–1):21	Physical address of the 2-MByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-18. Format of a 4-Level Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-17)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Table 4-19. Format of a 4-Level Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

.

63 62 61 60 59 58 57 56 55 54 53 52 51	M[1]	M-1	32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
Reserved[2]		Address of PML4 table	Ignored | PCD | PWT | Ign.	**CR3**
XD[3] | Ignored | Rsvd.		Address of page-directory-pointer table	Ign. | Rsvd | Ign | A | PCD | PWT | U/S | R/W | 1	**PML4E: present**
Ignored			0	**PML4E: not present**
XD | Prot. Key[4] | Ignored | Rsvd.		Address of 1GB page frame | Reserved	PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1	**PDPTE: 1GB page**
XD | Ignored | Rsvd.		Address of page directory	Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1	**PDPTE: page directory**
Ignored			0	**PDTPE: not present**
XD | Prot. Key[4] | Ignored | Rsvd.		Address of 2MB page frame | Reserved	PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1	**PDE: 2MB page**
XD | Ignored | Rsvd.		Address of page table	Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1	**PDE: page table**
Ignored			0	**PDE: not present**
XD | Prot. Key[4] | Ignored | Rsvd.		Address of 4KB page frame	Ign. | G | PAT | D | A | PCD | PWT | U/S | R/W | 1	**PTE: 4KB page**
Ignored			0	**PTE: not present**

Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging

NOTES:

1. M is an abbreviation for MAXPHYADDR.
2. Reserved fields must be 0.
3. If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.
4. If CR4.PKE = 0, the protection key is ignored.

4.6 ACCESS RIGHTS

There is a translation for a linear address if the processes described in Section 4.3, Section 4.4.2, and Section 4.5 (depending upon the paging mode) completes and produces a physical address. Whether an access is permitted by a translation is determined by the access rights specified by the paging-structure entries controlling the translation;[1] paging-mode modifiers in CR0, CR4, and the IA32_EFER MSR; EFLAGS.AC; and the mode of the access.

1. With PAE paging, the PDPTEs do not determine access rights.

Section 4.6.1 describes how the processor determines the access rights for each linear address. Section 4.6.2 provides additional information about how protection keys contribute to access-rights determination. (They do so only with 4-level paging and only if CR4.PKE = 1.)

4.6.1 Determination of Access Rights

Every access to a linear address is either a **supervisor-mode access** or a **user-mode access**. For all instruction fetches and most data accesses, this distinction is determined by the current privilege level (CPL): accesses made while CPL < 3 are supervisor-mode accesses, while accesses made while CPL = 3 are user-mode accesses.

Some operations implicitly access system data structures with linear addresses; the resulting accesses to those data structures are supervisor-mode accesses regardless of CPL. Examples of such accesses include the following: accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor; accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception; and accesses to the task-state segment (TSS) as part of a task switch or change of CPL. All these accesses are called **implicit supervisor-mode accesses** regardless of CPL. Other accesses made while CPL < 3 are called **explicit supervisor-mode accesses**.

Access rights are also controlled by the **mode** of a linear address as specified by the paging-structure entries controlling the translation of the linear address. If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the address is a **supervisor-mode address**. Otherwise, the address is a **user-mode address**.

The following items detail how paging determines access rights:

- For supervisor-mode accesses:
  - — Data may be read (implicitly or explicitly) from any supervisor-mode address.
  - — Data reads from user-mode pages.
    Access rights depend on the value of CR4.SMAP:
    - If CR4.SMAP = 0, data may be read from any user-mode address with a protection key for which read access is permitted.
    - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
      - — If EFLAGS.AC = 1 and the access is explicit, data may be read from any user-mode address with a protection key for which read access is permitted.
      - — If EFLAGS.AC = 0 or the access is implicit, data may not be read from any user-mode address.

    Section 4.6.2 explains how protection keys are associated with user-mode addresses and the accesses that are permitted for each protection key.
  - — Data writes to supervisor-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, data may be written to any supervisor-mode address.
    - If CR0.WP = 1, data may be written to any supervisor-mode address with a translation for which the R/W flag (bit 1) is 1 in every paging-structure entry controlling the translation; data may not be written to any supervisor-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
  - — Data writes to user-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, access rights depend on the value of CR4.SMAP:
      - — If CR4.SMAP = 0, data may be written to any user-mode address with a protection key for which write access is permitted.
      - — If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
        - If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a protection key for which write access is permitted.
        - If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.

- If CR0.WP = 1, access rights depend on the value of CR4.SMAP:
  - — If CR4.SMAP = 0, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
  - — If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
    - If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
    - If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.

Section 4.6.2 explains how protection keys are associated with user-mode addresses and the accesses that are permitted for each protection key.

— Instruction fetches from supervisor-mode addresses.

- For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any supervisor-mode address.
- For PAE paging or 4-level paging with IA32_EFER.NXE = 1, instructions may be fetched from any supervisor-mode address with a translation for which the XD flag (bit 63) is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any supervisor-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.

— Instruction fetches from user-mode addresses.
Access rights depend on the values of CR4.SMEP:

- If CR4.SMEP = 0, access rights depend on the paging mode and the value of IA32_EFER.NXE:
  - — For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
  - — For PAE paging or 4-level paging with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any user-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.
- If CR4.SMEP = 1, instructions may not be fetched from any user-mode address.

- For user-mode accesses:
  - — Data reads.
    Access rights depend on the mode of the linear address:
    - Data may be read from any user-mode address with a protection key for which read access is permitted. Section 4.6.2 explains how protection keys are associated with user-mode addresses and the accesses that are permitted for each protection key.
    - Data may not be read from any supervisor-mode address.
  - — Data writes.
    Access rights depend on the mode of the linear address:
    - Data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted. Section 4.6.2 explains how protection keys are associated with user-mode addresses and the accesses that are permitted for each protection key.
    - Data may not be written to any supervisor-mode address.
  - — Instruction fetches.
    Access rights depend on the mode of the linear address, the paging mode, and the value of IA32_EFER.NXE:

- For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
- For PAE paging or 4-level paging with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation.
- Instructions may not be fetched from any supervisor-mode address.

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). These structures may include information about access rights. The processor may enforce access rights based on the TLBs and paging-structure caches instead of on the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change access rights, the processor might not use that change for a subsequent access to an affected linear address (see Section 4.10.4.3). See Section 4.10.4.2 for how software can ensure that the processor uses the modified access rights.

4.6.2 Protection Keys

The protection-key feature provides an additional mechanism by which 4-level paging controls access to user-mode addresses. When CR4.PKE = 1, every linear address is associated with the 4-bit **protection key** located in bits 62:59 of the paging-structure entry that mapped the page containing the linear address (see Section 4.5). The PKRU register determines, for each protection key, whether user-mode addresses with that protection key may be read or written.

If CR4.PKE = 0, or if 4-level paging is not active, the processor does not associate linear addresses with protection keys and does not use the access-control mechanism described in this section. In either of these cases, a reference in Section 4.6.1 to a user-mode address with a protection key should be considered a reference to any user-mode address.

The PKRU register (protection key rights for user pages) is a 32-bit register with the following format: for each *i* (0 ≤ *i* ≤ 15), PKRU[2*i*] is the **access-disable bit** for protection key *i* (AD*i*); PKRU[2*i*+1] is the **write-disable bit** for protection key *i* (WD*i*).

Software can use the RDPKRU and WRPKRU instructions with ECX = 0 to read and write PKRU. In addition, the PKRU register is XSAVE-managed state and can thus be read and written by instructions in the XSAVE feature set. See Chapter 13, “Managing State Using the XSAVE Feature Set,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for more information about the XSAVE feature set.

How a linear address’s protection key controls access to the address depends on the mode of the linear address:

- A linear address’s protection key controls only data accesses to the address. It does not in any way affect instructions fetches from the address.
- The protection key of a supervisor-mode address is ignored and does not control data accesses to the address. Because of this, Section 4.6.1 does not refer to protection keys when specifying the access rights for supervisor-mode addresses.
- Use of the protection key *i* of a user-mode address depends on the value of the PKRU register:
  - If AD*i* = 1, no data accesses are permitted.
  - If WD*i* = 1, permission may be denied to certain data write accesses:
    - User-mode write accesses are not permitted.
    - Supervisor-mode write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, WD*i* does not affect supervisor-mode write accesses to user-mode addresses with protection key *i*.)

4.7 PAGE-FAULT EXCEPTIONS

Accesses using linear addresses may cause **page-fault exceptions** (#PF; exception 14). An access to a linear address may cause a page-fault exception for either of two reasons: (1) there is no translation for the linear address; or (2) there is a translation for the linear address, but its access rights do not permit the access.

As noted in Section 4.3, Section 4.4.2, and Section 4.5, there is no translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit. If there is a translation for a linear address, its access rights are determined as specified in Section 4.6.

When Intel® Software Guard Extensions (Intel® SGX) are enabled, the processor may deliver exception 14 for reasons unrelated to paging. See Section 37.3, "Access-control Requirements" and Section 37.19, "Enclave Page Cache Map (EPCM)" in Chapter 37, "Enclave Access Control and Data Structures." Such an exception is called an **SGX-induced page fault**. The processor uses the error code to distinguish SGX-induced page faults from ordinary page faults.

Figure 4-12 illustrates the error code that the processor provides on delivery of a page-fault exception. The following items explain how the bits in the error code describe the nature of the page-fault exception:

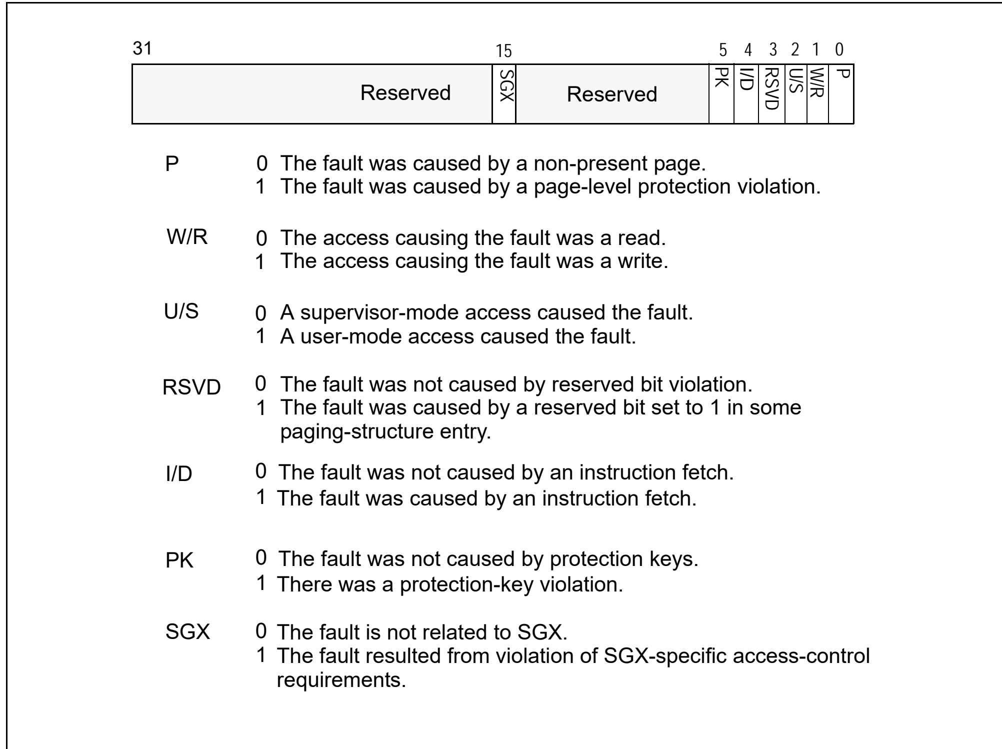


Figure 4-12. Page-Fault Error Code

- P flag (bit 0).
This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
- W/R (bit 1).
If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- U/S (bit 2).
If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging. User-mode and supervisor-mode accesses are defined in Section 4.6.

- RSVD flag (bit 3).
  This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address. (Because reserved bits are not checked in a paging-structure entry whose P flag is 0, bit 3 of the error code can be set only if bit 0 is also set.[1])

  Bits reserved in the paging-structure entries are reserved for future functionality. Software developers should be aware that such bits may be used in the future and that a paging-structure entry that causes a page-fault exception on one processor might not do so in the future.
- I/D flag (bit 4).
  This flag is 1 if (1) the access causing the page-fault exception was an instruction fetch; and (2) either (a) CR4.SMEP = 1; or (b) both (i) CR4.PAE = 1 (either PAE paging or 4-level paging is in use); and (ii) IA32_EFER.NXE = 1. Otherwise, the flag is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- PK flag (bit 5).
  This flag is 1 if (1) IA32_EFER.LMA = CR4.PKE = 1; (2) the access causing the page-fault exception was a data access; (3) the linear address was a user-mode address with protection key *i*; and (5) the PKRU register (see Section 4.6.2) is such that either (a) AD*i* = 1; or (b) the following all hold: (i) WD*i* = 1; (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access.
- SGX flag (bit 15).
  This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

Page-fault exceptions occur only due to an attempt to use a linear address. Failures to load the PDPTE registers with PAE paging (see Section 4.4.1) cause general-protection exceptions (#GP(0)) and not page-fault exceptions.

4.8 ACCESSED AND DIRTY FLAGS

For any paging-structure entry that is used during linear-address translation, bit 5 is the **accessed** flag.[2] For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the **dirty** flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are "sticky," meaning that, once set, the processor does not clear them; only software can clear them.

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected linear address (see Section 4.10.4.3). See Section 4.10.4.2 for how software can ensure that these bits are updated as desired.

NOTE

The accesses used by the processor to set these flags may or may not be exposed to the processor's self-modifying code detection logic. If the processor is executing code from the same

---

1. Some past processors had errata for some page faults that occur when there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address. Due to these errata, some such page faults produced error codes that cleared bit 0 (P flag) and set bit 3 (RSVD flag).

2. With PAE paging, the PDPTEs are not used during linear-address translation but only to load the PDPTE registers for some executions of the MOV CR instruction (see Section 4.4.1). For this reason, the PDPTEs do not contain accessed flags with PAE paging.

memory area that is being used for the paging structures, the setting of these flags may or may not result in an immediate change to the executing code stream.

4.9 PAGING AND MEMORY TYPING

The **memory type** of a memory access refers to the type of caching used for that access. Chapter 11, “Memory Cache Control” provides many details regarding memory typing in the Intel-64 and IA-32 architectures. This section describes how paging contributes to the determination of memory typing.

The way in which paging contributes to memory typing depends on whether the processor supports the **Page Attribute Table** (**PAT**; see Section 11.12).[1] Section 4.9.1 and Section 4.9.2 explain how paging contributes to memory typing depending on whether the PAT is supported.

4.9.1 Paging and Memory Typing When the PAT is Not Supported (Pentium Pro and Pentium II Processors)

NOTE

The PAT is supported on all processors that support 4-level paging. Thus, this section applies only to 32-bit paging and PAE paging.

If the PAT is not supported, paging contributes to memory typing in conjunction with the memory-type range registers (MTRRs) as specified in Table 11-6 in Section 11.5.2.1.

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a PCD value and a PWT value. The latter two values are determined as follows:

- For an access to a PDE with 32-bit paging, the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, the PCD and PWT values come from the relevant PDPTE register.
- For an access to a PTE, the PCD and PWT values come from the relevant PDE.
- For an access to the physical address that is the translation of a linear address, the PCD and PWT values come from the relevant PTE (if the translation uses a 4-KByte page) or the relevant PDE (otherwise).
- With PAE paging, the UC memory type is used when loading the PDPTEs (see Section 4.4.1).

4.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified in Table 11-7 in Section 11.5.2.2.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry *i* comprises bits 8*i*+7:8*i* of the MSR).

1. The PAT is supported on Pentium III and more recent processor families. See Section 4.1.4 for how to determine whether the PAT is supported.

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 11-11 in Section 11.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry *i* of the PAT, where *i* is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with 4-level paging):
  - For 4-level paging with CR4.PCIDE = 1, *i* = 0.
  - Otherwise, *i* = 2*PCD+PWT, where the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, *i* = 2*PCD+PWT, where the PCD and PWT values come from the relevant PDPTE register.
- For an access to a paging-structure entry X whose address is in another paging-structure entry Y, *i* = 2*PCD+PWT, where the PCD and PWT values come from Y.
- For an access to the physical address that is the translation of a linear address, *i* = 4*PAT+2*PCD+PWT, where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).
- With PAE paging, the WB memory type is used when loading the PDPTEs (see Section 4.4.1).[1]

4.9.3 Caching Paging-Related Information about Memory Typing

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). These structures may include information about memory typing. The processor may use memory-typing information from the TLBs and paging-structure caches instead of from the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change the memory-typing bits, the processor might not use that change for a subsequent translation using that entry or for access to an affected linear address. See Section 4.10.4.2 for how software can ensure that the processor uses the modified memory typing.

4.10 CACHING TRANSLATION INFORMATION

The Intel-64 and IA-32 architectures may accelerate the address-translation process by caching data from the paging structures on the processor. Because the processor does not ensure that the data that it caches are always consistent with the structures in memory, it is important for software developers to understand how and when the processor may cache such data. They should also understand what actions software can take to remove cached data that may be inconsistent and when it should do so. This section provides software developers information about the relevant processor operation.

Section 4.10.1 introduces process-context identifiers (PCIDs), which a logical processor may use to distinguish information cached for different linear-address spaces. Section 4.10.2 and Section 4.10.3 describe how the processor may cache information in translation lookaside buffers (TLBs) and paging-structure caches, respectively. Section 4.10.4 explains how software can remove inconsistent cached information by invalidating portions of the TLBs and paging-structure caches. Section 4.10.5 describes special considerations for multiprocessor systems.

4.10.1 Process-Context Identifiers (PCIDs)

Process-context identifiers (PCIDs) are a facility by which a logical processor may cache information for multiple linear-address spaces. The processor may retain cached information when software switches to a different linear-address space with a different PCID (e.g., by loading CR3; see Section 4.10.4.1 for details).

1. Some older IA-32 processors used the UC memory type when loading the PDPTEs. Some processors may use the UC memory type if CR0.CD = 1 or if the MTRRs are disabled. These behaviors are model-specific and not architectural.

A PCID is a 12-bit identifier. Non-zero PCIDs are enabled by setting the PCIDE flag (bit 17) of CR4. If CR4.PCIDE = 0, the current PCID is always 000H; otherwise, the current PCID is the value of bits 11:0 of CR3. Not all processors allow CR4.PCIDE to be set to 1; see Section 4.1.4 for how to determine whether this is allowed.

The processor ensures that CR4.PCIDE can be 1 only in IA-32e mode (thus, 32-bit paging and PAE paging use only PCID 000H). In addition, software can change CR4.PCIDE from 0 to 1 only if CR3[11:0] = 000H. These requirements are enforced by the following limitations on the MOV CR instruction:

- MOV to CR4 causes a general-protection exception (#GP) if it would change CR4.PCIDE from 0 to 1 and either IA32_EFER.LMA = 0 or CR3[11:0] ≠ 000H.
- MOV to CR0 causes a general-protection exception if it would clear CR0.PG to 0 while CR4.PCIDE = 1.

When a logical processor creates entries in the TLBs (Section 4.10.2) and paging-structure caches (Section 4.10.3), it associates those entries with the current PCID. When using entries in the TLBs and paging-structure caches to translate a linear address, a logical processor uses only those entries associated with the current PCID (see Section 4.10.2.4 for an exception).

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. This is because (1) if CR4.PCIDE = 0, the logical processor will associate any newly cached information with the current PCID, 000H; and (2) if MOV to CR4 clears CR4.PCIDE, all cached information is invalidated (see Section 4.10.4.1).

NOTE

In revisions of this manual that were produced when no processors allowed CR4.PCIDE to be set to 1, Section 4.10 discussed the caching of translation information without any reference to PCIDs. While the section now refers to PCIDs in its specification of this caching, this documentation change is not intended to imply any change to the behavior of processors that do not allow CR4.PCIDE to be set to 1.

4.10.2 Translation Lookaside Buffers (TLBs)

A processor may cache information about the translation of linear addresses in translation lookaside buffers (TLBs). In general, TLBs contain entries that map page numbers to page frames; these terms are defined in Section 4.10.2.1. Section 4.10.2.2 describes how information may be cached in TLBs, and Section 4.10.2.3 gives details of TLB usage. Section 4.10.2.4 explains the global-page feature, which allows software to indicate that certain translations should receive special treatment when cached in the TLBs.

4.10.2.1 Page Numbers, Page Frames, and Page Offsets

Section 4.3, Section 4.4.2, and Section 4.5 give details of how the different paging modes translate linear addresses to physical addresses. Specifically, the upper bits of a linear address (called the **page number**) determine the upper bits of the physical address (called the **page frame**); the lower bits of the linear address (called the **page offset**) determine the lower bits of the physical address. The boundary between the page number and the page offset is determined by the **page size**. Specifically:

- 32-bit paging:
  - If the translation does not use a PTE (because CR4.PSE = 1 and the PS flag is 1 in the PDE used), the page size is 4 MBytes and the page number comprises bits 31:22 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- PAE paging:
  - If the translation does not use a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 31:21 of the linear address.
  - If the translation does uses a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.

- 4-level paging:
  - If the translation does not use a PDE (because the PS flag is 1 in the PDPTE used), the page size is 1 GByte and the page number comprises bits 47:30 of the linear address.
  - If the translation does use a PDE but does not uses a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 47:21 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 47:12 of the linear address.

4.10.2.2 Caching Translations in TLBs

The processor may accelerate the paging process by caching individual translations in **translation lookaside buffers** (**TLBs**). Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number (see Section 4.6):
  - The logical-AND of the R/W flags.
  - The logical-AND of the U/S flags.
  - The logical-OR of the XD flags (necessary only if IA32_EFER.NXE = 1).
  - The protection key (necessary only with 4-level paging and CR4.PKE = 1).
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):
  - The dirty flag (see Section 4.8).
  - The memory type (see Section 4.9).

(TLB entries may contain other information as well. A processor may implement multiple TLBs, and some of these may be for special purposes, e.g., only for instruction fetches. Such special-purpose TLBs may not contain some of this information if it is not necessary. For example, a TLB used only for instruction fetches need not contain information about the R/W and dirty flags.)

As noted in Section 4.10.1, any TLB entries created by a logical processor are associated with the current PCID.

Processors need not implement any TLBs. Processors that do implement TLBs may invalidate any TLB entry at any time. Software should not rely on the existence of TLBs or on the retention of TLB entries.

4.10.2.3 Details of TLB Use

Because the TLBs cache entries only for linear addresses with translations, there can be a TLB entry for a page number only if the P flag is 1 and the reserved bits are 0 in each of the paging-structure entries used to translate that page number. In addition, the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation; before caching a translation, the processor sets any of these accessed flags that is not already 1.

The processor may cache translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path.

If the page number of a linear address corresponds to a TLB entry associated with the current PCID, the processor may use that TLB entry to determine the page frame, access rights, and other attributes for accesses to that linear address. In this case, the processor may not actually consult the paging structures in memory. The processor may retain a TLB entry unmodified even if software subsequently modifies the relevant paging-structure entries in memory. See Section 4.10.4.2 for how software can ensure that the processor uses the modified paging-structure entries.

If the paging structures specify a translation using a page larger than 4 KBytes, some processors may cache multiple smaller-page TLB entries for that translation. Each such TLB entry would be associated with a page

number corresponding to the smaller page size (e.g., bits 47:12 of a linear address with 4-level paging), even though part of that page number (e.g., bits 20:12) is part of the offset with respect to the page specified by the paging structures. The upper bits of the physical address in such a TLB entry are derived from the physical address in the PDE used to create the translation, while the lower bits come from the linear address of the access for which the translation is created. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. For example, an execution of INVLPG for a linear address on such a page invalidates any and all smaller-page TLB entries for the translation of any linear address on that page.

If software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes, the TLBs may subsequently contain multiple translations for the address range (one for each page size). A reference to a linear address in the address range may use any of these translations. Which translation is used may vary from one execution to another, and the choice may be implementation-specific.

4.10.2.4 Global Pages

The Intel-64 and IA-32 architectures also allow for **global pages** when the PGE flag (bit 7) is 1 in CR4. If the G flag (bit 8) is 1 in a paging-structure entry that maps a page (either a PTE or a paging-structure entry in which the PS flag is 1), any TLB entry cached for a linear address using that paging-structure entry is considered to be **global**. Because the G flag is used only in paging-structure entries that map a page, and because information from such entries is not cached in the paging-structure caches, the global-page feature does not affect the behavior of the paging-structure caches.

A logical processor may use a global TLB entry to translate a linear address, even if the TLB entry is associated with a PCID different from the current PCID.

4.10.3 Paging-Structure Caches

In addition to the TLBs, a processor may cache other information about the paging structures in memory.

4.10.3.1 Caches for Paging Structures

A processor may support any or all of the following paging-structure caches:

- **PML4 cache** (4-level paging only). Each PML4-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 47:39 have that value. The entry contains information from the PML4E used to translate such linear addresses:
  — The physical address from the PML4E (the address of the page-directory-pointer table).
  — The value of the R/W flag of the PML4E.
  — The value of the U/S flag of the PML4E.
  — The value of the XD flag of the PML4E.
  — The values of the PCD and PWT flags of the PML4E.

  The following items detail how a processor may use the PML4 cache:
  — If the processor has a PML4-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML4E in memory).
  — The processor does not create a PML4-cache entry unless the P flag is 1 and all reserved bits are 0 in the PML4E in memory.
  — The processor does not create a PML4-cache entry unless the accessed flag is 1 in the PML4E in memory; before caching a translation, the processor sets the accessed flag if it is not already 1.
  — The processor may create a PML4-cache entry even if there are no translations for any linear address that might use that entry (e.g., because the P flags are 0 in all entries in the referenced page-directory-pointer table).
  — If the processor creates a PML4-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E in memory.

- **PDPTE cache** (4-level paging only).[1] Each PDPTE-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 47:30 have that value. The entry contains information from the PML4E and PDPTE used to translate such linear addresses:
  - — The physical address from the PDPTE (the address of the page directory). (No PDPTE-cache entry is created for a PDPTE that maps a 1-GByte page.)
  - — The logical-AND of the R/W flags in the PML4E and the PDPTE.
  - — The logical-AND of the U/S flags in the PML4E and the PDPTE.
  - — The logical-OR of the XD flags in the PML4E and the PDPTE.
  - — The values of the PCD and PWT flags of the PDPTE.

  The following items detail how a processor may use the PDPTE cache:
  - — If the processor has a PDPTE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML4E and the PDPTE in memory).
  - — The processor does not create a PDPTE-cache entry unless the P flag is 1, the PS flag is 0, and the reserved bits are 0 in the PML4E and the PDPTE in memory.
  - — The processor does not create a PDPTE-cache entry unless the accessed flags are 1 in the PML4E and the PDPTE in memory; before caching a translation, the processor sets any accessed flags that are not already 1.
  - — The processor may create a PDPTE-cache entry even if there are no translations for any linear address that might use that entry.
  - — If the processor creates a PDPTE-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E or PDPTE in memory.
- **PDE cache**. The use of the PDE cache depends on the paging mode:
  - — For 32-bit paging, each PDE-cache entry is referenced by a 10-bit value and is used for linear addresses for which bits 31:22 have that value.
  - — For PAE paging, each PDE-cache entry is referenced by an 11-bit value and is used for linear addresses for which bits 31:21 have that value.
  - — For 4-level paging, each PDE-cache entry is referenced by a 27-bit value and is used for linear addresses for which bits 47:21 have that value.

  A PDE-cache entry contains information from the PML4E, PDPTE, and PDE used to translate the relevant linear addresses (for 32-bit paging and PAE paging, only the PDE applies):
  - — The physical address from the PDE (the address of the page table). (No PDE-cache entry is created for a PDE that maps a page.)
  - — The logical-AND of the R/W flags in the PML4E, PDPTE, and PDE.
  - — The logical-AND of the U/S flags in the PML4E, PDPTE, and PDE.
  - — The logical-OR of the XD flags in the PML4E, PDPTE, and PDE.
  - — The values of the PCD and PWT flags of the PDE.

1. With PAE paging, the PDPTEs are stored in internal, non-architectural registers. The operation of these registers is described in Section 4.4.1 and differs from that described here.

The following items detail how a processor may use the PDE cache (references below to PML4Es and PDPTEs apply only to 4-level paging):

— If the processor has a PDE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML4E, the PDPTE, and the PDE in memory).

— The processor does not create a PDE-cache entry unless the P flag is 1, the PS flag is 0, and the reserved bits are 0 in the PML4E, the PDPTE, and the PDE in memory.

— The processor does not create a PDE-cache entry unless the accessed flag is 1 in the PML4E, the PDPTE, and the PDE in memory; before caching a translation, the processor sets any accessed flags that are not already 1.

— The processor may create a PDE-cache entry even if there are no translations for any linear address that might use that entry.

— If the processor creates a PDE-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E, the PDPTE, or the PDE in memory.

Information from a paging-structure entry can be included in entries in the paging-structure caches for other paging-structure entries referenced by the original entry. For example, if the R/W flag is 0 in a PML4E, then the R/W flag will be 0 in any PDPTE-cache entry for a PDPTE from the page-directory-pointer table referenced by that PML4E. This is because the R/W flag of each such PDPTE-cache entry is the logical-AND of the R/W flags in the appropriate PML4E and PDPTE.

The paging-structure caches contain information only from paging-structure entries that reference other paging structures (and not those that map pages). Because the G flag is not used in such paging-structure entries, the global-page feature does not affect the behavior of the paging-structure caches.

The processor may create entries in paging-structure caches for translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path.

As noted in Section 4.10.1, any entries created in paging-structure caches by a logical processor are associated with the current PCID.

A processor may or may not implement any of the paging-structure caches. Software should rely on neither their presence nor their absence. The processor may invalidate entries in these caches at any time. Because the processor may create the cache entries at the time of translation and not update them following subsequent modifications to the paging structures in memory, software should take care to invalidate the cache entries appropriately when causing such modifications. The invalidation of TLBs and the paging-structure caches is described in Section 4.10.4.

4.10.3.2 Using the Paging-Structure Caches to Translate Linear Addresses

When a linear address is accessed, the processor uses a procedure such as the following to determine the physical address to which it translates and whether the access should be allowed:

- If the processor finds a TLB entry that is for the page number of the linear address and that is associated with the current PCID (or which is global), it may use the physical address, access rights, and other attributes from that entry.
- If the processor does not find a relevant TLB entry, it may use the upper bits of the linear address to select an entry from the PDE cache that is associated with the current PCID (Section 4.10.3.1 indicates which bits are used in each paging mode). It can then use that entry to complete the translation process (locating a PTE, etc.) as if it had traversed the PDE (and, for 4-level paging, the PDPTE and PML4E) corresponding to the PDE-cache entry.
- The following items apply when 4-level paging is used:

  — If the processor does not find a relevant TLB entry or a relevant PDE-cache entry, it may use bits 47:30 of the linear address to select an entry from the PDPTE cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PDE, etc.) as if it had traversed the PDPTE and the PML4E corresponding to the PDPTE-cache entry.

  — If the processor does not find a relevant TLB entry, a relevant PDE-cache entry, or a relevant PDPTE-cache entry, it may use bits 47:39 of the linear address to select an entry from the PML4 cache that is associated

with the current PCID. It can then use that entry to complete the translation process (locating a PDPTE, etc.) as if it had traversed the corresponding PML4E.

(Any of the above steps would be skipped if the processor does not support the cache in question.)

If the processor does not find a TLB or paging-structure-cache entry for the linear address, it uses the linear address to traverse the entire paging-structure hierarchy, as described in Section 4.3, Section 4.4.2, and Section 4.5.

4.10.3.3 Multiple Cached Entries for a Single Paging-Structure Entry

The paging-structure caches and TLBs may contain multiple entries associated with a single PCID and with information derived from a single paging-structure entry. The following items give some examples for 4-level paging:

- Suppose that two PML4Es contain the same physical address and thus reference the same page-directory-pointer table. Any PDPTE in that table may result in two PDPTE-cache entries, each associated with a different set of linear addresses. Specifically, suppose that the n_1^th and n_2^th entries in the PML4 table contain the same physical address. This implies that the physical address in the mth PDPTE in the page-directory-pointer table would appear in the PDPTE-cache entries associated with both p_1 and p_2, where (p_1 » 9) = n_1, (p_2 » 9) = n_2, and (p_1 & 1FFH) = (p_2 & 1FFH) = m. This is because both PDPTE-cache entries use the same PDPTE, one resulting from a reference from the n_1^th PML4E and one from the n_2^th PML4E.
- Suppose that the first PML4E (i.e., the one in position 0) contains the physical address X in CR3 (the physical address of the PML4 table). This implies the following:
  - Any PML4-cache entry associated with linear addresses with 0 in bits 47:39 contains address X.
  - Any PDPTE-cache entry associated with linear addresses with 0 in bits 47:30 contains address X. This is because the translation for a linear address for which the value of bits 47:30 is 0 uses the value of bits 47:39 (0) to locate a page-directory-pointer table at address X (the address of the PML4 table). It then uses the value of bits 38:30 (also 0) to find address X again and to store that address in the PDPTE-cache entry.
  - Any PDE-cache entry associated with linear addresses with 0 in bits 47:21 contains address X for similar reasons.
  - Any TLB entry for page number 0 (associated with linear addresses with 0 in bits 47:12) translates to page frame X » 12 for similar reasons.

  The same PML4E contributes its address X to all these cache entries because the self-referencing nature of the entry causes it to be used as a PML4E, a PDPTE, a PDE, and a PTE.

4.10.4 Invalidation of TLBs and Paging-Structure Caches

As noted in Section 4.10.2 and Section 4.10.3, the processor may create entries in the TLBs and the paging-structure caches when linear addresses are translated, and it may retain these entries even after the paging structures used to create them have been modified. To ensure that linear-address translation uses the modified paging structures, software should take action to invalidate any cached entries that may contain information that has since been modified.

4.10.4.1 Operations that Invalidate TLBs and Paging-Structure Caches

The following instructions invalidate entries in the TLBs and the paging-structure caches:

- INVLPG. This instruction takes a single operand, which is a linear address. The instruction invalidates any TLB entries that are for a page number corresponding to the linear address and that are associated with the current PCID. It also invalidates any global TLB entries with that page number, regardless of PCID (see Section 4.10.2.4).[1] INVLPG also invalidates all entries in all paging-structure caches associated with the current PCID, regardless of the linear addresses to which they correspond.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3), the instruction invalidates all of them.

- INVPCID. The operation of this instruction is based on instruction operands, called the INVPCID type and the INVPCID descriptor. Four INVPCID types are currently defined:
  — Individual-address. If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—associated with the PCID specified in the INVPCID descriptor and that would be used to translate the linear address specified in the INVPCID descriptor.[1] (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs and for other linear addresses.)
  — Single-context. If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs.)
  — All-context, including globals. If the INVPCID type is 2, the logical processor invalidates mappings—including global translations—associated with all PCIDs.
  — All-context. If the INVPCID type is 3, the logical processor invalidates mappings—except global translations—associated with all PCIDs. (The instruction may also invalidate global translations.)

  See Chapter 3 of the *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 2A* for details of the INVPCID instruction.
- MOV to CR0. The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if it changes the value of CR0.PG from 1 to 0.
- MOV to CR3. The behavior of the instruction depends on the value of CR4.PCIDE:
  — If CR4.PCIDE = 0, the instruction invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.
  — If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 0, the instruction invalidates all TLB entries associated with the PCID specified in bits 11:0 of the instruction's source operand except those for global pages. It also invalidates all entries in all paging-structure caches associated with that PCID. It is not required to invalidate entries in the TLBs and paging-structure caches that are associated with other PCIDs.
  — If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 1, the instruction is not required to invalidate any TLB entries or entries in paging-structure caches.
- MOV to CR4. The behavior of the instruction depends on the bits being modified:
  — The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if (1) it changes the value of CR4.PGE;[2] or (2) it changes the value of the CR4.PCIDE from 1 to 0.
  — The instruction invalidates all TLB entries and all entries in all paging-structure caches for the current PCID if (1) it changes the value of CR4.PAE; or (2) it changes the value of CR4.SMEP from 0 to 1.
- Task switch. If a task switch changes the value of CR3, it invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.[3]
- VMX transitions. See Section 4.11.1.

The processor is always free to invalidate additional entries in the TLBs and paging-structure caches. The following are some examples:

- INVLPG may invalidate TLB entries for pages other than the one corresponding to its linear-address operand. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the current PCID.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3), the instruction invalidates all of them.
2. If CR4.PGE is changing from 0 to 1, there were no global TLB entries before the execution; if CR4.PGE is changing from 1 to 0, there will be no global TLB entries after the execution.
3. Task switches do not occur in IA-32e mode and thus cannot occur with 4-level paging. Since CR4.PCIDE can be set only with 4-level paging, task switches occur only with CR4.PCIDE = 0.

- INVPCID may invalidate TLB entries for pages other than the one corresponding to the specified linear address. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the specified PCID.
- MOV to CR0 may invalidate TLB entries even if CR0.PG is not changing. For example, this may occur if either CR0.CD or CR0.NW is modified.
- MOV to CR3 may invalidate TLB entries for global pages. If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 0, it may invalidate TLB entries and entries in the paging-structure caches associated with PCIDs other than the PCID it is establishing. It may invalidate entries if CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 1.
- MOV to CR4 may invalidate TLB entries when changing CR4.PSE or when changing CR4.SMEP from 1 to 0.
- On a processor supporting Hyper-Threading Technology, invalidations performed on one logical processor may invalidate entries in the TLBs and paging-structure caches used by other logical processors.

(Other instructions and operations may invalidate entries in the TLBs and the paging-structure caches, but the instructions identified above are recommended.)

In addition to the instructions identified above, page faults invalidate entries in the TLBs and paging-structure caches. In particular, a page-fault exception resulting from an attempt to use a linear address will invalidate any TLB entries that are for a page number corresponding to that linear address and that are associated with the current PCID. It also invalidates all entries in the paging-structure caches that would be used for that linear address and that are associated with the current PCID.[1] These invalidations ensure that the page-fault exception will not recur (if the faulting instruction is re-executed) if it would not be caused by the contents of the paging structures in memory (and if, therefore, it resulted from cached entries that were not invalidated after the paging structures were modified in memory).

As noted in Section 4.10.2, some processors may choose to cache multiple smaller-page TLB entries for a translation specified by the paging structures to use a page larger than 4 KBytes. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. The INVLPG instruction and page faults provide the same assurances that they provide when a single TLB entry is used: they invalidate all TLB entries corresponding to the translation specified by the paging structures.

4.10.4.2 Recommended Invalidation

The following items provide some recommendations regarding when software should perform invalidations:

- If software modifies a paging-structure entry that maps a page (rather than referencing another paging structure), it should execute INVLPG for any linear address with a page number whose translation uses that paging-structure entry.[2]

  (If the paging-structure entry may be used in the translation of different page numbers — see Section 4.10.3.3 — software should execute INVLPG for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)
- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:
  - Execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLPG at least once.
  - Execute MOV to CR3 if the modified entry controls no global pages.
  - Execute MOV to CR4 to modify CR4.PGE.
- If CR4.PCIDE = 1 and software modifies a paging-structure entry that does not map a page or in which the G flag (bit 8) is 0, additional steps are required if the entry may be used for PCIDs other than the current one. Any one of the following suffices:

1. Unlike INVLPG, page faults need not invalidate **all** entries in the paging-structure caches, only those that would be used to translate the faulting linear address.
2. One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes.

— Execute MOV to CR4 to modify CR4.PGE, either immediately or before again using any of the affected PCIDs. For example, software could use different (previously unused) PCIDs for the processes that used the affected PCIDs.

— For each affected PCID, execute MOV to CR3 to make that PCID current (and to load the address of the appropriate PML4 table). If the modified entry controls no global pages and bit 63 of the source operand to MOV to CR3 was 0, no further steps are required. Otherwise, execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry; if no page numbers that would use the entry have translations, execute INVLPG at least once.

- If software using PAE paging modifies a PDPTE, it should reload CR3 with the register's current value to ensure that the modified PDPTE is loaded into the corresponding PDPTE register (see Section 4.4.1).
- If the nature of the paging structures is such that a single entry may be used for multiple purposes (see Section 4.10.3.3), software should perform invalidations for all of these purposes. For example, if a single entry might serve as both a PDE and PTE, it may be necessary to execute INVLPG with two (or more) linear addresses, one that uses the entry as a PDE and one that uses it as a PTE. (Alternatively, software could use MOV to CR3 or MOV to CR4.)
- As noted in Section 4.10.2, the TLBs may subsequently contain multiple translations for the address range if software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes. A reference to a linear address in the address range may use any of these translations.

  Software wishing to prevent this uncertainty should not write to a paging-structure entry in a way that would change, for any linear address, both the page size and either the page frame, access rights, or other attributes. It can instead use the following algorithm: first clear the P flag in the relevant paging-structure entry (e.g., PDE); then invalidate any translations for the affected linear addresses (see above); and then modify the relevant paging-structure entry to set the P flag and establish modified translation(s) for the new page size.
- Software should clear bit 63 of the source operand to a MOV to CR3 instruction that establishes a PCID that had been used earlier for a different linear-address space (e.g., with a different value in bits 51:12 of CR3). This ensures invalidation of any information that may have been cached for the previous linear-address space.

  This assumes that both linear-address spaces use the same global pages and that it is thus not necessary to invalidate any global TLB entries. If that is not the case, software should invalidate those entries by executing MOV to CR4 to modify CR4.PGE.

4.10.4.3 Optional Invalidation

The following items describe cases in which software may choose not to invalidate and the potential consequences of that choice:

- If a paging-structure entry is modified to change the P flag from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the P flag is 0.[1]
- If a paging-structure entry is modified to change the accessed flag from 0 to 1, no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the accessed flag is 0.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).
- If CR4.SMEP = 0 and a paging-structure entry is modified to change the U/S flag from 0 to 1, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted user-mode access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).
- If a paging-structure entry is modified to change the XD flag from 1 to 0, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted instruction fetch) but no other

1. If it is also the case that no invalidation was performed the last time the P flag was changed from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the P flag had earlier been 1.

adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).

- If a paging-structure entry is modified to change the accessed flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent access to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such an access has not occurred.
- If software modifies a paging-structure entry that identifies the final physical address for a linear address (either a PTE or a paging-structure entry in which the PS flag is 1) to change the dirty flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent write to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such a write has not occurred.
- The read of a paging-structure entry in translating an address being used to fetch an instruction may appear to execute before an earlier write to that paging-structure entry if there is no serializing instruction between the write and the instruction fetch. Note that the invalidating instructions identified in Section 4.10.4.1 are all serializing instructions.
- Section 4.10.3.3 describes situations in which a single paging-structure entry may contain information cached in multiple entries in the paging-structure caches. Because all entries in these caches are invalidated by any execution of INVLPG, it is not necessary to follow the modification of such a paging-structure entry by executing INVLPG multiple times solely for the purpose of invalidating these multiple cached entries. (It may be necessary to do so to invalidate multiple TLB entries.)

4.10.4.4 Delayed Invalidation

Required invalidations may be delayed under some circumstances. Software developers should understand that, between the modification of a paging-structure entry and execution of the invalidation instruction recommended in Section 4.10.4.2, the processor may use translations based on either the old value or the new value of the paging-structure entry. The following items describe some of the potential consequences of delayed invalidation:

- If a paging-structure entry is modified to change the P flag from 1 to 0, an access to a linear address whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, write accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the U/S flag from 0 to 1, user-mode accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the XD flag from 1 to 0, instruction fetches from linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.

As noted in Section 8.1.1, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory and invalidation has been delayed, some of the accesses may complete (writing to memory) while another causes a page-fault exception.[1] In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault.

In some cases, the consequences of delayed invalidation may not affect software adversely. For example, when freeing a portion of the linear-address space (by marking paging-structure entries “not present”), invalidation using INVLPG may be delayed if software does not re-allocate that portion of the linear-address space or the memory that had been associated with it. However, because of speculative execution (or errant software), there may be accesses to the freed portion of the linear-address space before the invalidations occur. In this case, the following can happen:

- Reads can occur to the freed portion of the linear-address space. Therefore, invalidation should not be delayed for an address range that has read side effects.
- The processor may retain entries in the TLBs and paging-structure caches for an extended period of time. Software should not assume that the processor will not use entries associated with a linear address simply because time has passed.

---

1. If the accesses are to different pages, this may occur even if invalidation has not been delayed.

- As noted in Section 4.10.3.1, the processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry. Thus, if software has marked "not present" all entries in a page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table (assuming that the PDE itself is marked "present").
- If software attempts to write to the freed portion of the linear-address space, the processor might not generate a page fault. (Such an attempt would likely be the result of a software error.) For that reason, the page frames previously associated with the freed portion of the linear-address space should not be reallocated for another purpose until the appropriate invalidations have been performed.

4.10.5 Propagation of Paging-Structure Changes to Multiple Processors

As noted in Section 4.10.4, software that modifies a paging-structure entry may need to invalidate entries in the TLBs and paging-structure caches that were derived from the modified entry before it was modified. In a system containing more than one logical processor, software must account for the fact that there may be entries in the TLBs and paging-structure caches of logical processors other than the one used to modify the paging-structure entry. The process of propagating the changes to a paging-structure entry is commonly referred to as "TLB shoot-down."

TLB shootdown can be done using memory-based semaphores and/or interprocessor interrupts (IPI). The following items describe a simple but inefficient example of a TLB shootdown algorithm for processors supporting the Intel-64 and IA-32 architectures:

1. Begin barrier: Stop all but one logical processor; that is, cause all but one to execute the HLT instruction or to enter a spin loop.
2. Allow the active logical processor to change the necessary paging-structure entries.
3. Allow all logical processors to perform invalidations appropriate to the modifications to the paging-structure entries.
4. Allow all logical processors to resume normal operation.

Alternative, performance-optimized, TLB shootdown algorithms may be developed; however, software developers must take care to ensure that the following conditions are met:

- All logical processors that are using the paging structures that are being modified must participate and perform appropriate invalidations after the modifications are made.
- If the modifications to the paging-structure entries are made before the barrier or if there is no barrier, the operating system must ensure one of the following: (1) that the affected linear-address range is not used between the time of modification and the time of invalidation; or (2) that it is prepared to deal with the consequences of the affected linear-address range being used during that period. For example, if the operating system does not allow pages being freed to be reallocated for another purpose until after the required invalidations, writes to those pages by errant software will not unexpectedly modify memory that is in use.
- Software must be prepared to deal with reads, instruction fetches, and prefetch requests to the affected linear-address range that are a result of speculative execution that would never actually occur in the executed code path.

When multiple logical processors are using the same linear-address space at the same time, they must coordinate before any request to modify the paging-structure entries that control that linear-address space. In these cases, the barrier in the TLB shootdown routine may not be required. For example, when freeing a range of linear addresses, some other mechanism can assure no logical processor is using that range before the request to free it is made. In this case, a logical processor freeing the range can clear the P flags in the PTEs associated with the range, free the physical page frames associated with the range, and then signal the other logical processors using that linear-address space to perform the necessary invalidations. All the affected logical processors must complete their invalidations before the linear-address range and the physical page frames previously associated with that range can be reallocated.

4.11 INTERACTIONS WITH VIRTUAL-MACHINE EXTENSIONS (VMX)

The architecture for virtual-machine extensions (VMX) includes features that interact with paging. Section 4.11.1 discusses ways in which VMX-specific control transfers, called VMX transitions specially affect paging. Section 4.11.2 gives an overview of VMX features specifically designed to support address translation.

4.11.1 VMX Transitions

The VMX architecture defines two control transfers called **VM entries** and **VM exits**; collectively, these are called **VMX transitions**. VM entries and VM exits are described in detail in Chapter 26 and Chapter 27, respectively, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. The following items identify paging-related details:

- VMX transitions modify the CR0 and CR4 registers and the IA32_EFER MSR concurrently. For this reason, they allow transitions between paging modes that would not otherwise be possible:
  - VM entries allow transitions from 4-level paging directly to either 32-bit paging or PAE paging.
  - VM exits allow transitions from either 32-bit paging or PAE paging directly to 4-level paging.
- VMX transitions that result in PAE paging load the PDPTE registers (see Section 4.4.1) as follows:
  - VM entries load the PDPTE registers either from the physical address being loaded into CR3 or from the virtual-machine control structure (VMCS); see Section 26.3.2.4.
  - VM exits load the PDPTE registers from the physical address being loaded into CR3; see Section 27.5.4.
- VMX transitions invalidate the TLBs and paging-structure caches based on certain control settings. See Section 26.3.2.5 and Section 27.5.5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

4.11.2 VMX Support for Address Translation

Chapter 28, "VMX Support for Address Translation," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* describe two features of the virtual-machine extensions (VMX) that interact directly with paging. These are **virtual-processor identifiers** (**VPIDs**) and the **extended page table** mechanism (**EPT**).

VPIDs provide a way for software to identify to the processor the address spaces for different "virtual processors." The processor may use this identification to maintain concurrently information for multiple address spaces in its TLBs and paging-structure caches, even when non-zero PCIDs are not being used. See Section 28.1 for details.

When EPT is in use, the addresses in the paging-structures are not used as physical addresses to access memory and memory-mapped I/O. Instead, they are treated as **guest-physical** addresses and are translated through a set of EPT paging structures to produce physical addresses. EPT can also specify its own access rights and memory typing; these are used on conjunction with those specified in this chapter. See Section 28.2 for more information.

Both VPIDs and EPT may change the way that a processor maintains information in TLBs and paging structure caches and the ways in which software can manage that information. Some of the behaviors documented in Section 4.10 may change. See Section 28.3 for details.

4.12 USING PAGING FOR VIRTUAL MEMORY

With paging, portions of the linear-address space need not be mapped to the physical-address space; data for the unmapped addresses can be stored externally (e.g., on disk). This method of mapping the linear-address space is referred to as virtual memory or demand-paged virtual memory.

Paging divides the linear address space into fixed-size pages that can be mapped into the physical-address space and/or external storage. When a program (or task) references a linear address, the processor uses paging to translate the linear address into a corresponding physical address if such an address is defined.

If the page containing the linear address is not currently mapped into the physical-address space, the processor generates a page-fault exception as described in Section 4.7. The handler for page-fault exceptions typically

directs the operating system or executive to load data for the unmapped page from external storage into physical memory (perhaps writing a different page from physical memory out to external storage in the process) and to map it using paging (by updating the paging structures). When the page has been loaded into physical memory, a return from the exception handler causes the instruction that generated the exception to be restarted.

Paging differs from segmentation through its use of fixed-size pages. Unlike segments, which usually are the same size as the code or data structures they hold, pages have a fixed size. If segmentation is the only form of address translation used, a data structure present in physical memory will have all of its parts in memory. If paging is used, a data structure can be partly in memory and partly in disk storage.

4.13 MAPPING SEGMENTS TO PAGES

The segmentation and paging mechanisms provide support for a wide variety of approaches to memory management. When segmentation and paging are combined, segments can be mapped to pages in several ways. To implement a flat (unsegmented) addressing environment, for example, all the code, data, and stack modules can be mapped to one or more large segments (up to 4-GBytes) that share same range of linear addresses (see Figure 3-2 in Section 3.2.2). Here, segments are essentially invisible to applications and the operating-system or executive. If paging is used, the paging mechanism can map a single linear-address space (contained in a single segment) into virtual memory. Alternatively, each program (or task) can have its own large linear-address space (contained in its own segment), which is mapped into virtual memory through its own paging structures.

Segments can be smaller than the size of a page. If one of these segments is placed in a page which is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-Byte semaphore, occupies 4 KBytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.

The Intel-64 and IA-32 architectures do not enforce correspondence between the boundaries of pages and segments. A page can contain the end of one segment and the beginning of another. Similarly, a segment can contain the end of one page and the beginning of another.

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment which can fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

One approach to combining paging and segmentation that simplifies memory-management software is to give each segment its own page table, as shown in Figure 4-13. This convention gives the segment a single entry in the page directory, and this entry provides the access control information for paging the entire segment.

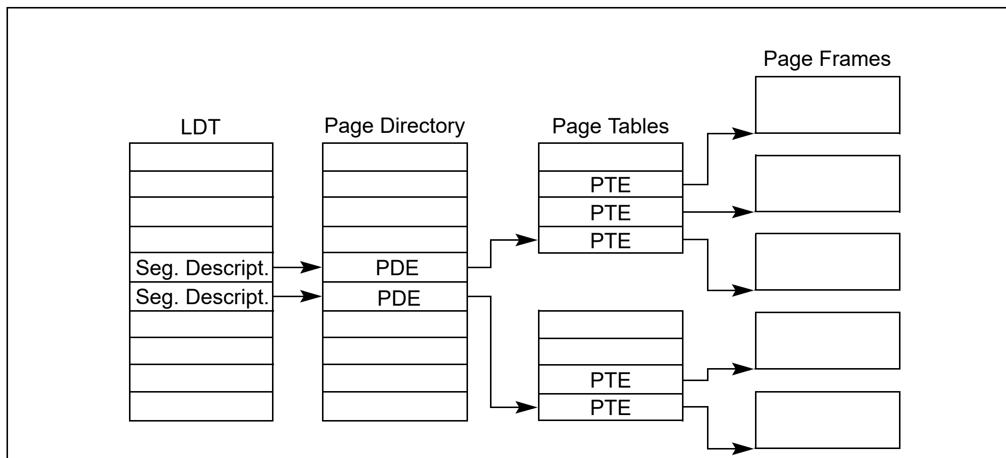


Figure 4-13. Memory Management Convention That Assigns a Page Table to Each Segment

10. Updates to Chapter 5, Volume 3A

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Change to this chapter: Updated term "IA-32e paging" to "4-level paging"; included footnote on first usage of new term.

CHAPTER 5
PROTECTION

In protected mode, the Intel 64 and IA-32 architectures provide a protection mechanism that operates at both the segment level and the page level. This protection mechanism provides the ability to limit access to certain segments or pages based on privilege levels (four privilege levels for segments and two privilege levels for pages). For example, critical operating-system code and data can be protected by placing them in more privileged segments than those that contain applications code. The processor's protection mechanism will then prevent application code from accessing the operating-system code and data in any but a controlled, defined manner.

Segment and page protection can be used at all stages of software development to assist in localizing and detecting design problems and bugs. It can also be incorporated into end-products to offer added robustness to operating systems, utilities software, and applications software.

When the protection mechanism is used, each memory reference is checked to verify that it satisfies various protection checks. All checks are made before the memory cycle is started; any violation results in an exception. Because checks are performed in parallel with address translation, there is no performance penalty. The protection checks that are performed fall into the following categories:

- Limit checks.
- Type checks.
- Privilege level checks.
- Restriction of addressable domain.
- Restriction of procedure entry-points.
- Restriction of instruction set.

All protection violation results in an exception being generated. See Chapter 6, "Interrupt and Exception Handling," for an explanation of the exception mechanism. This chapter describes the protection mechanism and the violations which lead to exceptions.

The following sections describe the protection mechanism available in protected mode. See Chapter 20, "8086 Emulation," for information on protection in real-address and virtual-8086 mode.

5.1 ENABLING AND DISABLING SEGMENT AND PAGE PROTECTION

Setting the PE flag in register CR0 causes the processor to switch to protected mode, which in turn enables the segment-protection mechanism. Once in protected mode, there is no control bit for turning the protection mechanism on or off. The part of the segment-protection mechanism that is based on privilege levels can essentially be disabled while still in protected mode by assigning a privilege level of 0 (most privileged) to all segment selectors and segment descriptors. This action disables the privilege level protection barriers between segments, but other protection checks such as limit checking and type checking are still carried out.

Page-level protection is automatically enabled when paging is enabled (by setting the PG flag in register CR0). Here again there is no mode bit for turning off page-level protection once paging is enabled. However, page-level protection can be disabled by performing the following operations:

- Clear the WP flag in control register CR0.
- Set the read/write (R/W) and user/supervisor (U/S) flags for each page-directory and page-table entry.

This action makes each page a writable, user page, which in effect disables page-level protection.

5.2 FIELDS AND FLAGS USED FOR SEGMENT-LEVEL AND PAGE-LEVEL PROTECTION

The processor's protection mechanism uses the following fields and flags in the system data structures to control access to segments and pages:

- **Descriptor type (S) flag —** (Bit 12 in the second doubleword of a segment descriptor.) Determines if the segment descriptor is for a system segment or a code or data segment.
- **Type field** — (Bits 8 through 11 in the second doubleword of a segment descriptor.) Determines the type of code, data, or system segment.
- **Limit field** — (Bits 0 through 15 of the first doubleword and bits 16 through 19 of the second doubleword of a segment descriptor.) Determines the size of the segment, along with the G flag and E flag (for data segments).
- **G flag —** (Bit 23 in the second doubleword of a segment descriptor.) Determines the size of the segment, along with the limit field and E flag (for data segments).
- **E flag —** (Bit 10 in the second doubleword of a data-segment descriptor.) Determines the size of the segment, along with the limit field and G flag.
- **Descriptor privilege level (DPL) field** — (Bits 13 and 14 in the second doubleword of a segment descriptor.) Determines the privilege level of the segment.
- **Requested privilege level (RPL) field** — (Bits 0 and 1 of any segment selector.) Specifies the requested privilege level of a segment selector.
- **Current privilege level (CPL) field** — (Bits 0 and 1 of the CS segment register.) Indicates the privilege level of the currently executing program or procedure. The term current privilege level (CPL) refers to the setting of this field.
- **User/supervisor (U/S) flag** — (Bit 2 of paging-structure entries.) Determines the type of page: user or supervisor.
- **Read/write (R/W) flag** — (Bit 1 of paging-structure entries.) Determines the type of access allowed to a page: read-only or read/write.
- **Execute-disable (XD) flag** — (Bit 63 of certain paging-structure entries.) Determines the type of access allowed to a page: executable or not-executable.

Figure 5-1 shows the location of the various fields and flags in the data-, code-, and system-segment descriptors; Figure 3-6 shows the location of the RPL (or CPL) field in a segment selector (or the CS register); and Chapter 4 identifies the locations of the U/S, R/W, and XD flags in the paging-structure entries.

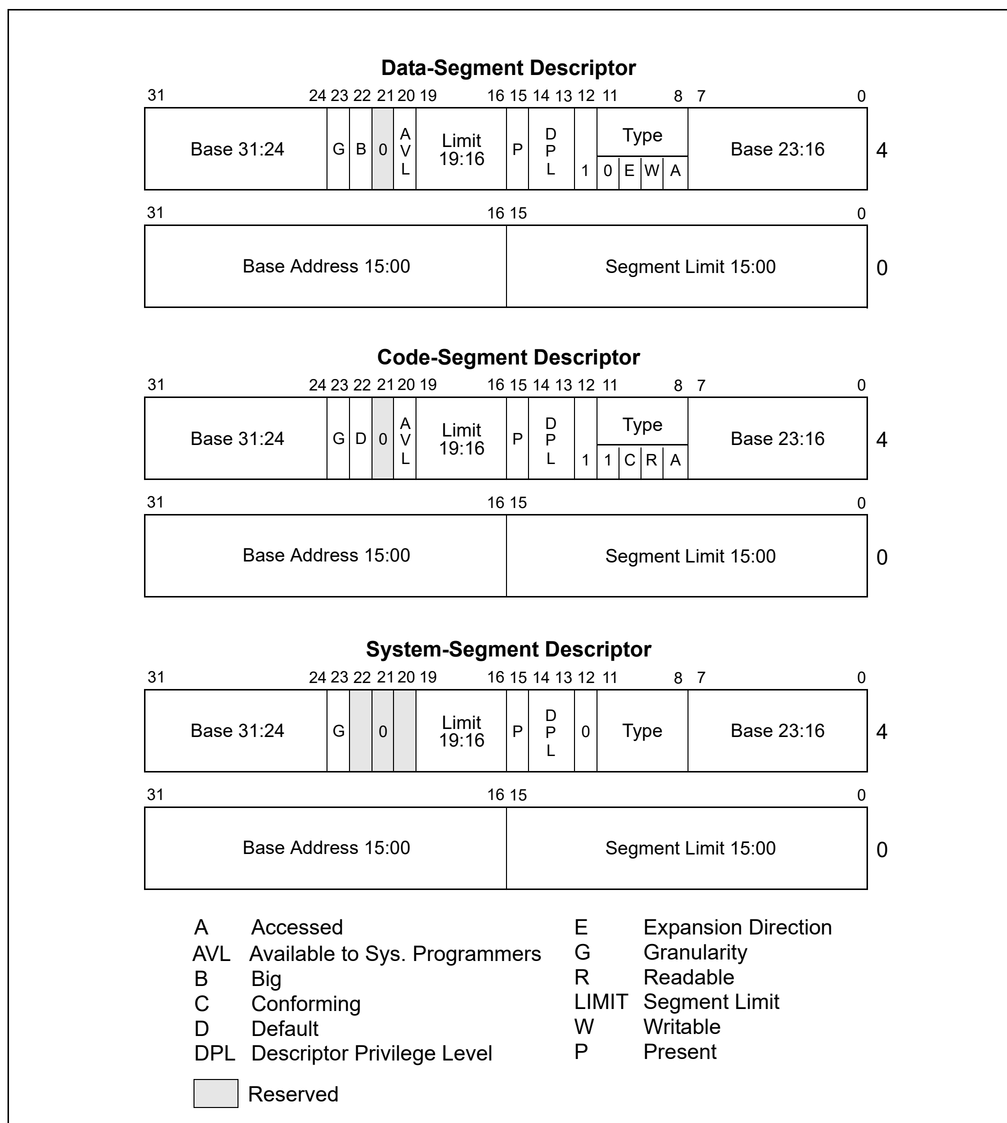


Figure 5-1. Descriptor Fields Used for Protection

Many different styles of protection schemes can be implemented with these fields and flags. When the operating system creates a descriptor, it places values in these fields and flags in keeping with the particular protection style chosen for an operating system or executive. Application programs do not generally access or modify these fields and flags.

The following sections describe how the processor uses these fields and flags to perform the various categories of checks described in the introduction to this chapter.

5.2.1 Code-Segment Descriptor in 64-bit Mode

Code segments continue to exist in 64-bit mode even though, for address calculations, the segment base is treated as zero. Some code-segment (CS) descriptor content (the base address and limit fields) is ignored; the remaining fields function normally (except for the readable bit in the type field).

Code segment descriptors and selectors are needed in IA-32e mode to establish the processor's operating mode and execution privilege-level. The usage is as follows:

- IA-32e mode uses a previously unused bit in the CS descriptor. Bit 53 is defined as the 64-bit (L) flag and is used to select between 64-bit mode and compatibility mode when IA-32e mode is active (IA32_EFER.LMA = 1). See Figure 5-2.
  - If CS.L = 0 and IA-32e mode is active, the processor is running in compatibility mode. In this case, CS.D selects the default size for data and addresses. If CS.D = 0, the default data and address size is 16 bits. If CS.D = 1, the default data and address size is 32 bits.
  - If CS.L = 1 and IA-32e mode is active, the only valid setting is CS.D = 0. This setting indicates a default operand size of 32 bits and a default address size of 64 bits. The CS.L = 1 and CS.D = 1 bit combination is reserved for future use and a #GP fault will be generated on an attempt to use a code segment with these bits set in IA-32e mode.
- In IA-32e mode, the CS descriptor's DPL is used for execution privilege checks (as in legacy 32-bit mode).

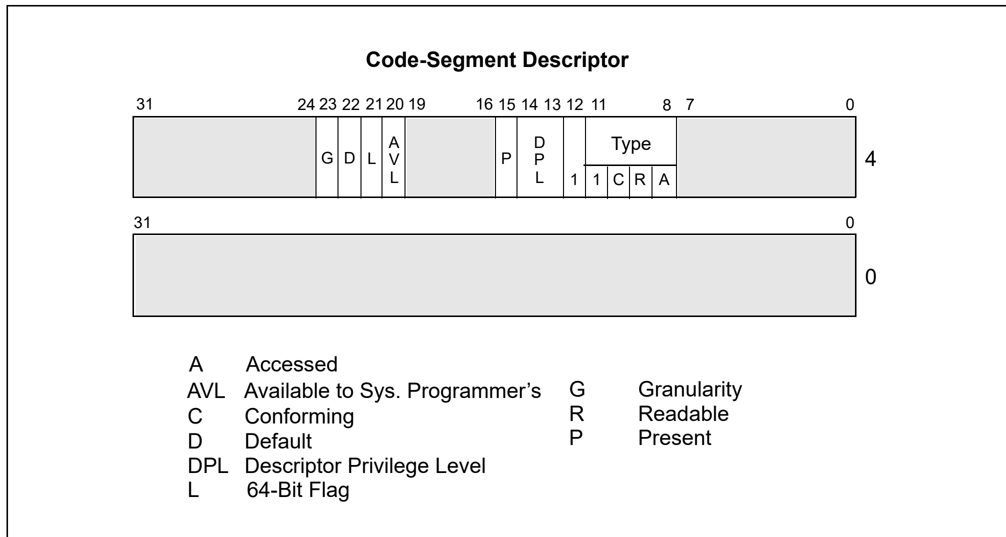


Figure 5-2. Descriptor Fields with Flags used in IA-32e Mode

5.3 LIMIT CHECKING

The limit field of a segment descriptor prevents programs or procedures from addressing memory locations outside the segment. The effective value of the limit depends on the setting of the G (granularity) flag (see Figure 5-1). For data segments, the limit also depends on the E (expansion direction) flag and the B (default stack pointer size and/or upper bound) flag. The E flag is one of the bits in the type field when the segment descriptor is for a data-segment type.

When the G flag is clear (byte granularity), the effective limit is the value of the 20-bit limit field in the segment descriptor. Here, the limit ranges from 0 to FFFFFH (1 MByte). When the G flag is set (4-KByte page granularity), the processor scales the value in the limit field by a factor of 2^{12} (4 KBytes). In this case, the effective limit ranges from FFFH (4 KBytes) to FFFFFFFFH (4 GBytes). Note that when scaling is used (G flag is set), the lower 12 bits of a segment offset (address) are not checked against the limit; for example, note that if the segment limit is 0, offsets 0 through FFFH are still valid.

For all types of segments except expand-down data segments, the effective limit is the last address that is allowed to be accessed in the segment, which is one less than the size, in bytes, of the segment. The processor causes a general-protection exception (or, if the segment is SS, a stack-fault exception) any time an attempt is made to access the following addresses in a segment:

- A byte at an offset greater than the effective limit
- A word at an offset greater than the (effective-limit – 1)
- A doubleword at an offset greater than the (effective-limit – 3)
- A quadword at an offset greater than the (effective-limit – 7)

- A double quadword at an offset greater than the (effective limit – 15)

When the effective limit is FFFFFFFFH (4 GBytes), these accesses may or may not cause the indicated exceptions. Behavior is implementation-specific and may vary from one execution to another.

For expand-down data segments, the segment limit has the same function but is interpreted differently. Here, the effective limit specifies the last address that is not allowed to be accessed within the segment; the range of valid offsets is from (effective-limit + 1) to FFFFFFFFH if the B flag is set and from (effective-limit + 1) to FFFFH if the B flag is clear. An expand-down segment has maximum size when the segment limit is 0.

Limit checking catches programming errors such as runaway code, runaway subscripts, and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier. Without limit checking, these errors could overwrite code or data in another segment.

In addition to checking segment limits, the processor also checks descriptor table limits. The GDTR and IDTR registers contain 16-bit limit values that the processor uses to prevent programs from selecting a segment descriptors outside the respective descriptor tables. The LDTR and task registers contain 32-bit segment limit value (read from the segment descriptors for the current LDT and TSS, respectively). The processor uses these segment limits to prevent accesses beyond the bounds of the current LDT and TSS. See Section 3.5.1, “Segment Descriptor Tables,” for more information on the GDT and LDT limit fields; see Section 6.10, “Interrupt Descriptor Table (IDT),” for more information on the IDT limit field; and see Section 7.2.4, “Task Register,” for more information on the TSS segment limit field.

5.3.1 Limit Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime limit checking on code or data segments. However, the processor does check descriptor-table limits.

5.4 TYPE CHECKING

Segment descriptors contain type information in two places:

- The S (descriptor type) flag.
- The type field.

The processor uses this information to detect programming errors that result in an attempt to use a segment or gate in an incorrect or unintended manner.

The S flag indicates whether a descriptor is a system type or a code or data type. The type field provides 4 additional bits for use in defining various types of code, data, and system descriptors. Table 3-1 shows the encoding of the type field for code and data descriptors; Table 3-2 shows the encoding of the field for system descriptors.

The processor examines type information at various times while operating on segment selectors and segment descriptors. The following list gives examples of typical operations where type checking is performed (this list is not exhaustive):

- **When a segment selector is loaded into a segment register** — Certain segment registers can contain only certain descriptor types, for example:
  — The CS register only can be loaded with a selector for a code segment.
  — Segment selectors for code segments that are not readable or for system segments cannot be loaded into data-segment registers (DS, ES, FS, and GS).
  — Only segment selectors of writable data segments can be loaded into the SS register.
- When a segment selector is loaded into the LDTR or task register — For example:
  — The LDTR can only be loaded with a selector for an LDT.
  — The task register can only be loaded with a segment selector for a TSS.
- **When instructions access segments whose descriptors are already loaded into segment registers** — Certain segments can be used by instructions only in certain predefined ways, for example:
  — No instruction may write into an executable segment.

- No instruction may write into a data segment if it is not writable.
- No instruction may read an executable segment unless the readable flag is set.

- **When an instruction operand contains a segment selector** — Certain instructions can access segments or gates of only a particular type, for example:
  - A far CALL or far JMP instruction can only access a segment descriptor for a conforming code segment, nonconforming code segment, call gate, task gate, or TSS.
  - The LLDT instruction must reference a segment descriptor for an LDT.
  - The LTR instruction must reference a segment descriptor for a TSS.
  - The LAR instruction must reference a segment or gate descriptor for an LDT, TSS, call gate, task gate, code segment, or data segment.
  - The LSL instruction must reference a segment descriptor for a LDT, TSS, code segment, or data segment.
  - IDT entries must be interrupt, trap, or task gates.
- **During certain internal operations** — For example:
  - On a far call or far jump (executed with a far CALL or far JMP instruction), the processor determines the type of control transfer to be carried out (call or jump to another code segment, a call or jump through a gate, or a task switch) by checking the type field in the segment (or gate) descriptor pointed to by the segment (or gate) selector given as an operand in the CALL or JMP instruction. If the descriptor type is for a code segment or call gate, a call or jump to another code segment is indicated; if the descriptor type is for a TSS or task gate, a task switch is indicated.
  - On a call or jump through a call gate (or on an interrupt- or exception-handler call through a trap or interrupt gate), the processor automatically checks that the segment descriptor being pointed to by the gate is for a code segment.
  - On a call or jump to a new task through a task gate (or on an interrupt- or exception-handler call to a new task through a task gate), the processor automatically checks that the segment descriptor being pointed to by the task gate is for a TSS.
  - On a call or jump to a new task by a direct reference to a TSS, the processor automatically checks that the segment descriptor being pointed to by the CALL or JMP instruction is for a TSS.
  - On return from a nested task (initiated by an IRET instruction), the processor checks that the previous task link field in the current TSS points to a TSS.

5.4.1 Null Segment Selector Checking

Attempting to load a null segment selector (see Section 3.4.2, "Segment Selectors") into the CS or SS segment register generates a general-protection exception (#GP). A null segment selector can be loaded into the DS, ES, FS, or GS register, but any attempt to access a segment through one of these registers when it is loaded with a null segment selector results in a #GP exception being generated. Loading unused data-segment registers with a null segment selector is a useful method of detecting accesses to unused segment registers and/or preventing unwanted accesses to data segments.

5.4.1.1 NULL Segment Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime checking on NULL segment selectors. The processor does not cause a #GP fault when an attempt is made to access memory where the referenced segment register has a NULL segment selector.

5.5 PRIVILEGE LEVELS

The processor's segment-protection mechanism recognizes 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. Figure 5-3 shows how these levels of privilege can be interpreted as rings of protection.

The center (reserved for the most privileged code, data, and stacks) is used for the segments containing the critical software, usually the kernel of an operating system. Outer rings are used for less critical software. (Systems that use only 2 of the 4 possible privilege levels should use levels 0 and 3.)

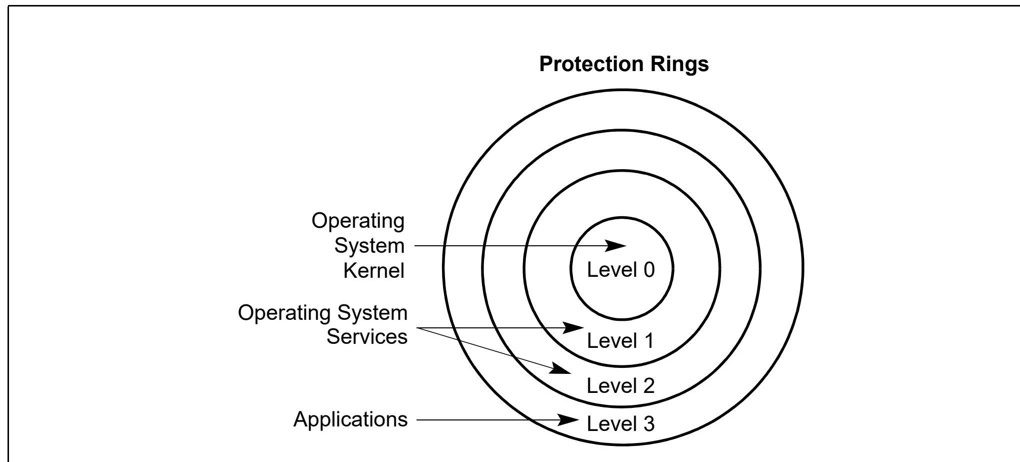


Figure 5-3. Protection Rings

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater privilege, except under controlled situations. When the processor detects a privilege level violation, it generates a general-protection exception (#GP).

To carry out privilege-level checks between code segments and data segments, the processor recognizes the following three types of privilege levels:

- **Current privilege level (CPL)** — The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.
- **Descriptor privilege level (DPL)** — The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector (as described later in this section). The DPL is interpreted differently, depending on the type of segment or gate being accessed:
  - **Data segment** — The DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
  - **Nonconforming code segment (without using a call gate)** — The DPL indicates the privilege level that a program or task must be at to access the segment. For example, if the DPL of a nonconforming code segment is 0, only programs running at a CPL of 0 can access the segment.
  - **Call gate** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the call gate. (This is the same access rule as for a data segment.)
  - **Conforming code segment and nonconforming code segment accessed through a call gate** — The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.

— **TSS** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the TSS. (This is the same access rule as for a data segment.)

- **Requested privilege level (RPL)** — The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector. The processor checks the RPL along with the CPL to determine if access to a segment is allowed. Even if the program or task requesting access to a segment has sufficient privilege to access the segment, access is denied if the RPL is not of sufficient privilege level. That is, if the RPL of a segment selector is numerically greater than the CPL, the RPL overrides the CPL, and vice versa. The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment. See Section 5.10.4, "Checking Caller Access Privileges (ARPL Instruction)," for a detailed description of the purpose and typical use of the RPL.

Privilege levels are checked when the segment selector of a segment descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of program control among code segments; therefore, the two kinds of accesses are considered separately in the following sections.

5.6 PRIVILEGE LEVEL CHECKING WHEN ACCESSING DATA SEGMENTS

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS). (Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (see Figure 5-4) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment's segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a general-protection fault is generated and the segment register is not loaded.

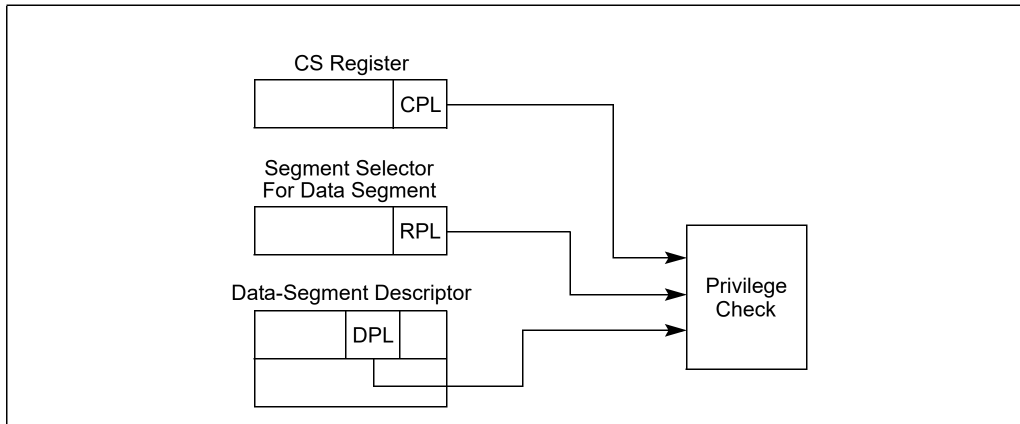


Figure 5-4. Privilege Check for Data Access

Figure 5-5 shows four procedures (located in codes segments A, B, C, and D), each running at different privilege levels and each attempting to access the same data segment.

1. The procedure in code segment A is able to access data segment E using segment selector E1, because the CPL of code segment A and the RPL of segment selector E1 are equal to the DPL of data segment E.
2. The procedure in code segment B is able to access data segment E using segment selector E2, because the CPL of code segment B and the RPL of segment selector E2 are both numerically lower than (more privileged) than the DPL of data segment E. A code segment B procedure can also access data segment E using segment selector E1.
3. The procedure in code segment C is not able to access data segment E using segment selector E3 (dotted line), because the CPL of code segment C and the RPL of segment selector E3 are both numerically greater than (less privileged) than the DPL of data segment E. Even if a code segment C procedure were to use segment selector

E1 or E2, such that the RPL would be acceptable, it still could not access data segment E because its CPL is not privileged enough.

4. The procedure in code segment D should be able to access data segment E because code segment D's CPL is numerically less than the DPL of data segment E. However, the RPL of segment selector E3 (which the code segment D procedure is using to access data segment E) is numerically greater than the DPL of data segment E, so access is not allowed. If the code segment D procedure were to use segment selector E1 or E2 to access the data segment, access would be allowed.

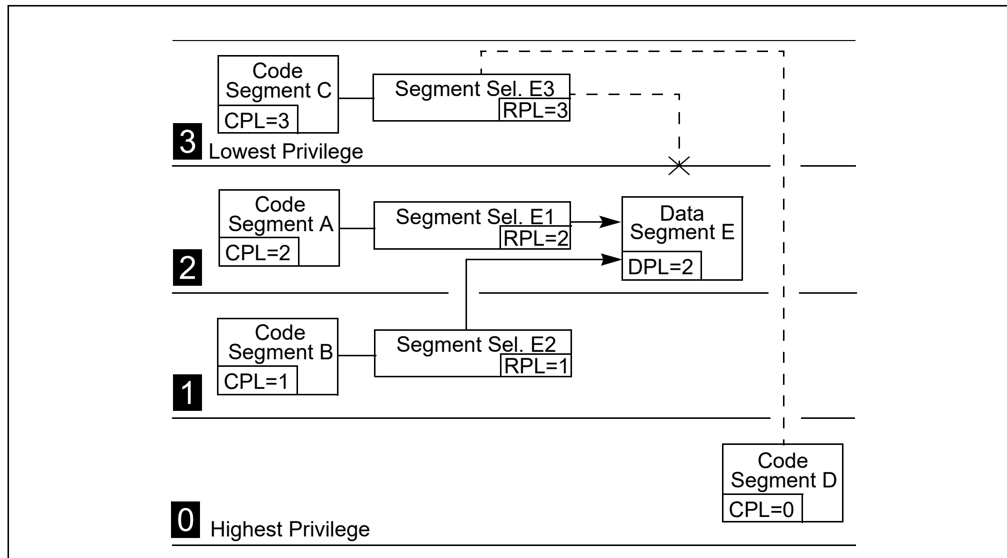


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

As demonstrated in the previous examples, the addressable domain of a program or task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

The RPL of a segment selector can always override the addressable domain of a program or task. When properly used, RPLs can prevent problems caused by accidental (or intensional) use of segment selectors for privileged data segments by less privileged programs or procedures.

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a data- segment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 5.10.4, "Checking Caller Access Privileges (ARPL Instruction)").

5.6.1 Accessing Data in Code Segments

In some instances it may be desirable to access data structures that are contained in a code segment. The following methods of accessing data in code segments are possible:

- Load a data-segment register with a segment selector for a nonconforming, readable, code segment.
- Load a data-segment register with a segment selector for a conforming, readable, code segment.
- Use a code-segment override prefix (CS) to read a readable, code segment whose selector is already loaded in the CS register.

The same rules for accessing data segments apply to method 1. Method 2 is always valid because the privilege level of a conforming code segment is effectively the same as the CPL, regardless of its DPL. Method 3 is always valid because the DPL of the code segment selected by the CS register is the same as the CPL.

5.7 PRIVILEGE LEVEL CHECKING WHEN LOADING THE SS REGISTER

Privilege level checking also occurs when the SS register is loaded with the segment selector for a stack segment. Here all privilege levels related to the stack segment must match the CPL; that is, the CPL, the RPL of the stack-segment selector, and the DPL of the stack-segment descriptor must be the same. If the RPL and DPL are not equal to the CPL, a general-protection exception (#GP) is generated.

5.8 PRIVILEGE LEVEL CHECKING WHEN TRANSFERRING PROGRAM CONTROL BETWEEN CODE SEGMENTS

To transfer program control from one code segment to another, the segment selector for the destination code segment must be loaded into the code-segment register (CS). As part of this loading process, the processor examines the segment descriptor for the destination code segment and performs various limit, type, and privilege checks. If these checks are successful, the CS register is loaded, program control is transferred to the new code segment, and program execution begins at the instruction pointed to by the EIP register.

Program control transfers are carried out with the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, SYSRET, INT *n*, and IRET instructions, as well as by the exception and interrupt mechanisms. Exceptions, interrupts, and the IRET instruction are special cases discussed in Chapter 6, “Interrupt and Exception Handling.” This chapter discusses only the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, and SYSRET instructions.

A JMP or CALL instruction can reference another code segment in any of four ways:

- The target operand contains the segment selector for the target code segment.
- The target operand points to a call-gate descriptor, which contains the segment selector for the target code segment.
- The target operand points to a TSS, which contains the segment selector for the target code segment.
- The target operand points to a task gate, which points to a TSS, which in turn contains the segment selector for the target code segment.

The following sections describe first two types of references. See Section 7.3, “Task Switching,” for information on transferring program control through a task gate and/or TSS.

The SYSENTER and SYSEXIT instructions are special instructions for making fast calls to and returns from operating system or executive procedures. These instructions are discussed in Section 5.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”

The SYCALL and SYSRET instructions are special instructions for making fast calls to and returns from operating system or executive procedures in 64-bit mode. These instructions are discussed in Section 5.8.8, “Fast System Calls in 64-Bit Mode.”

5.8.1 Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, so privilege-level checks are not performed. The far forms of the JMP, CALL, and RET instructions transfer control to other code segments, so the processor does perform privilege-level checks.

When transferring program control to another code segment without going through a call gate, the processor examines four kinds of privilege level and type information (see Figure 5-6):

- The CPL. (Here, the CPL is the privilege level of the calling code segment; that is, the code segment that contains the procedure that is making the call or jump.)

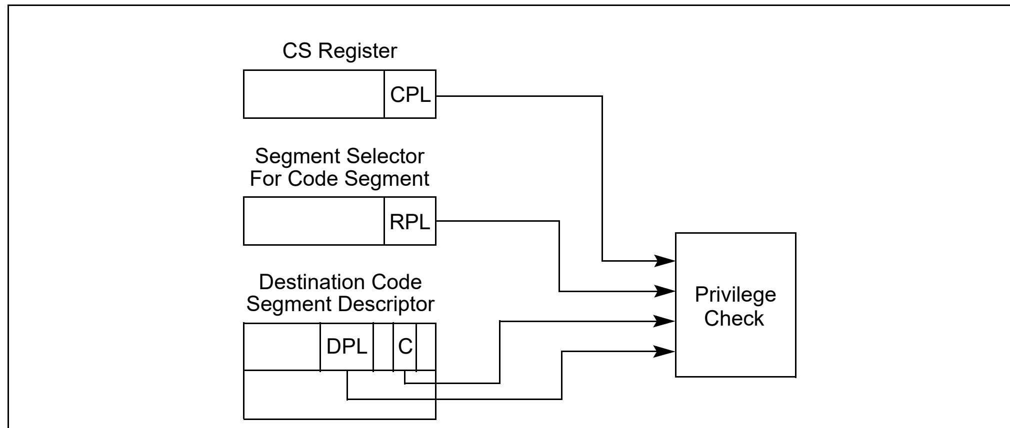


Figure 5-6. Privilege Check for Control Transfer Without Using a Gate

- The DPL of the segment descriptor for the destination code segment that contains the called procedure.
- The RPL of the segment selector of the destination code segment.
- The conforming (C) flag in the segment descriptor for the destination code segment, which determines whether the segment is a conforming (C flag is set) or nonconforming (C flag is clear) code segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types," for more information about this flag.

The rules that the processor uses to check the CPL, RPL, and DPL depends on the setting of the C flag, as described in the following sections.

5.8.1.1 Accessing Nonconforming Code Segments

When accessing nonconforming code segments, the CPL of the calling procedure must be equal to the DPL of the destination code segment; otherwise, the processor generates a general-protection exception (#GP). For example in Figure 5-7:

- Code segment C is a nonconforming code segment. A procedure in code segment A can call a procedure in code segment C (using segment selector C1) because they are at the same privilege level (CPL of code segment A is equal to the DPL of code segment C).
- A procedure in code segment B cannot call a procedure in code segment C (using segment selector C2 or C1) because the two code segments are at different privilege levels.

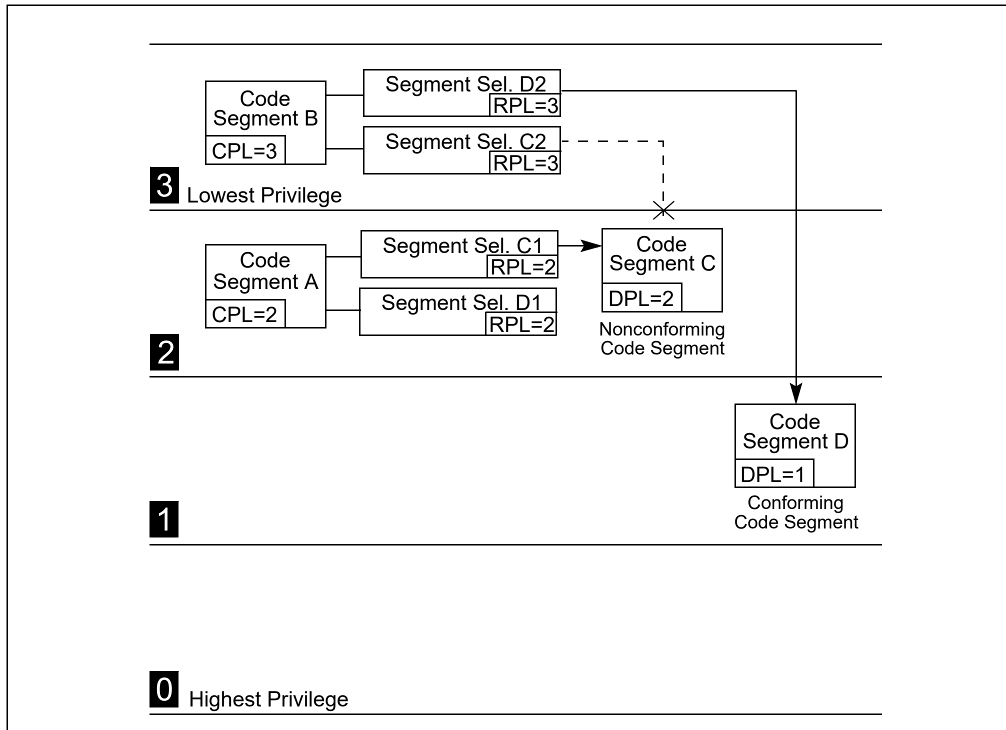


Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

The RPL of the segment selector that points to a nonconforming code segment has a limited effect on the privilege check. The RPL must be numerically less than or equal to the CPL of the calling procedure for a successful control transfer to occur. So, in the example in Figure 5-7, the RPLs of segment selectors C1 and C2 could legally be set to 0, 1, or 2, but not to 3.

When the segment selector of a nonconforming code segment is loaded into the CS register, the privilege level field is not changed; that is, it remains at the CPL (which is the privilege level of the calling procedure). This is true, even if the RPL of the segment selector is different from the CPL.

5.8.1.2 Accessing Conforming Code Segments

When accessing conforming code segments, the CPL of the calling procedure may be numerically equal to or greater than (less privileged) the DPL of the destination code segment; the processor generates a general-protection exception (#GP) only if the CPL is less than the DPL. (The segment selector RPL for the destination code segment is not checked if the segment is a conforming code segment.)

In the example in Figure 5-7, code segment D is a conforming code segment. Therefore, calling procedures in both code segment A and B can access code segment D (using either segment selector D1 or D2, respectively), because they both have CPLs that are greater than or equal to the DPL of the conforming code segment. **For conforming code segments, the DPL represents the numerically lowest privilege level that a calling procedure may be at to successfully make a call to the code segment**.

(Note that segments selectors D1 and D2 are identical except for their respective RPLs. But since RPLs are not checked when accessing conforming code segments, the two segment selectors are essentially interchangeable.)

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is less than the CPL. This situation is the only one where the CPL may be different from the DPL of the current code segment. Also, since the CPL does not change, no stack switch occurs.

Conforming segments are used for code modules such as math libraries and exception handlers, which support applications but do not require access to protected system facilities. These modules are part of the operating system or executive software, but they can be executed at numerically higher privilege levels (less privileged levels). Keeping the CPL at the level of a calling code segment when switching to a conforming code segment

prevents an application program from accessing nonconforming code segments while at the privilege level (DPL) of a conforming code segment and thus prevents it from accessing more privileged data.

Most code segments are nonconforming. For these segments, program control can be transferred only to code segments at the same level of privilege, unless the transfer is carried out through a call gate, as described in the following sections.

5.8.2 Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

Task gates are used for task switching and are discussed in Chapter 7, "Task Management". Trap and interrupt gates are special kinds of call gates used for calling exception and interrupt handlers. The are described in Chapter 6, "Interrupt and Exception Handling." This chapter is concerned only with call gates.

5.8.3 Call Gates

Call gates facilitate controlled transfers of program control between different privilege levels. They are typically used only in operating systems or executives that use the privilege-level protection mechanism. Call gates are also useful for transferring program control between 16-bit and 32-bit code segments, as described in Section 21.4, "Transferring Control Among Mixed-Size Code Segments."

Figure 5-8 shows the format of a call-gate descriptor. A call-gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT). It performs six functions:

- It specifies the code segment to be accessed.
- It defines an entry point for a procedure in the specified code segment.
- It specifies the privilege level required for a caller trying to access the procedure.

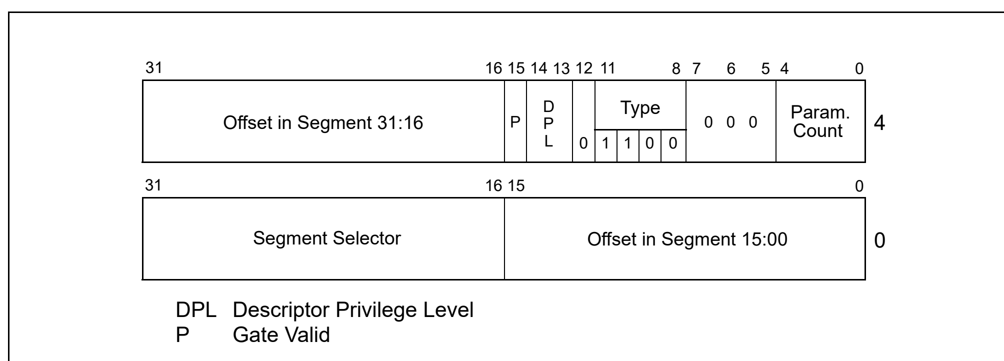


Figure 5-8. Call-Gate Descriptor

- If a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.
- It defines the size of values to be pushed onto the target stack: 16-bit gates force 16-bit pushes and 32-bit gates force 32-bit pushes.
- It specifies whether the call-gate descriptor is valid.

The segment selector field in a call gate specifies the code segment to be accessed. The offset field specifies the entry point in the code segment. This entry point is generally to the first instruction of a specific procedure. The DPL field indicates the privilege level of the call gate, which in turn is the privilege level required to access the

selected procedure through the gate. The P flag indicates whether the call-gate descriptor is valid. (The presence of the code segment to which the gate points is indicated by the P flag in the code segment's descriptor.) The parameter count field indicates the number of parameters to copy from the calling procedures stack to the new stack if a stack switch occurs (see Section 5.8.5, "Stack Switching"). The parameter count specifies the number of words for 16-bit call gates and doublewords for 32-bit call gates.

Note that the P flag in a gate descriptor is normally always set to 1. If it is set to 0, a not present (#NP) exception is generated when a program attempts to access the descriptor. The operating system can use the P flag for special purposes. For example, it could be used to track the number of times the gate is used. Here, the P flag is initially set to 0 causing a trap to the not-present exception handler. The exception handler then increments a counter and sets the P flag to 1, so that on returning from the handler, the gate descriptor will be valid.

5.8.3.1 IA-32e Mode Call Gates

Call-gate descriptors in 32-bit mode provide a 32-bit offset for the instruction pointer (EIP); 64-bit extensions double the size of 32-bit mode call gates in order to store 64-bit instruction pointers (RIP). See Figure 5-9:

- The first eight bytes (bytes 7:0) of a 64-bit mode call gate are similar but not identical to legacy 32-bit mode call gates. The parameter-copy-count field has been removed.
- Bytes 11:8 hold the upper 32 bits of the target-segment offset in canonical form. A general-protection exception (#GP) is generated if software attempts to use a call gate with a target offset that is not in canonical form.
- 16-byte descriptors may reside in the same descriptor table with 16-bit and 32-bit descriptors. A type field, used for consistency checking, is defined in bits 12:8 of the 64-bit descriptor's highest dword (cleared to zero). A general-protection exception (#GP) results if an attempt is made to access the upper half of a 64-bit mode descriptor as a 32-bit mode descriptor.

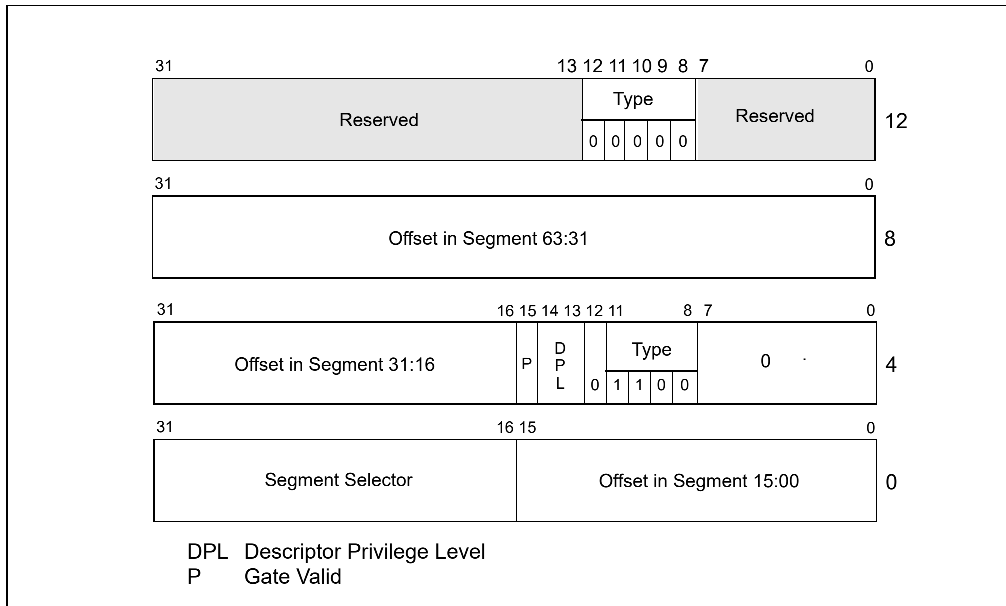


Figure 5-9. Call-Gate Descriptor in IA-32e Mode

- Target code segments referenced by a 64-bit call gate must be 64-bit code segments (CS.L = 1, CS.D = 0). If not, the reference generates a general-protection exception, #GP (CS selector).
- Only 64-bit mode call gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). The legacy 32-bit mode call gate type (0CH) is redefined in IA-32e mode as a 64-bit call-gate type; no 32-bit call-gate type exists in IA-32e mode.

- If a far call references a 16-bit call gate type (04H) in IA-32e mode, a general-protection exception (#GP) is generated.

When a call references a 64-bit mode call gate, actions taken are identical to those taken in 32-bit mode, with the following exceptions:

- Stack pushes are made in eight-byte increments.
- A 64-bit RIP is pushed onto the stack.
- Parameter copying is not performed.

Use a matching far-return instruction size for correct operation (returns from 64-bit calls must be performed with a 64-bit operand-size return to process the stack correctly).

5.8.4 Accessing a Code Segment Through a Call Gate

To access a call gate, a far pointer to the gate is provided as a target operand in a CALL or JMP instruction. The segment selector from this pointer identifies the call gate (see Figure 5-10); the offset from the pointer is required, but not used or checked by the processor. (The offset can be set to any value.)

When the processor has accessed the call gate, it uses the segment selector from the call gate to locate the segment descriptor for the destination code segment. (This segment descriptor can be in the GDT or the LDT.) It then combines the base address from the code-segment descriptor with the offset from the call gate to form the linear address of the procedure entry point in the code segment.

As shown in Figure 5-11, four different privilege levels are used to check the validity of a program control transfer through a call gate:

- The CPL (current privilege level).
- The RPL (requestor's privilege level) of the call gate’s selector.
- The DPL (descriptor privilege level) of the call gate descriptor.
- The DPL of the segment descriptor of the destination code segment.

The C flag (conforming) in the segment descriptor for the destination code segment is also checked.

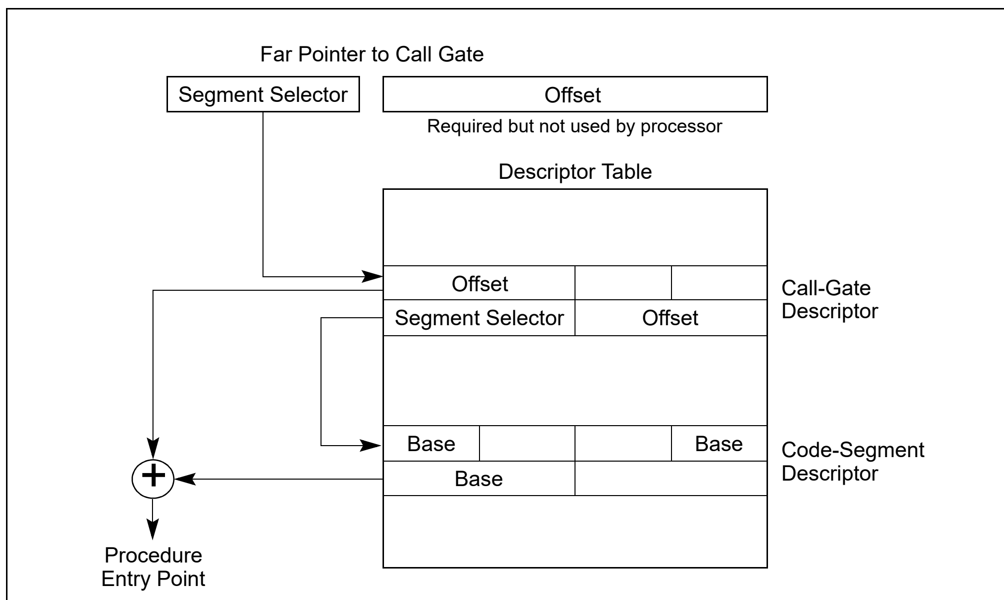


Figure 5-10. Call-Gate Mechanism

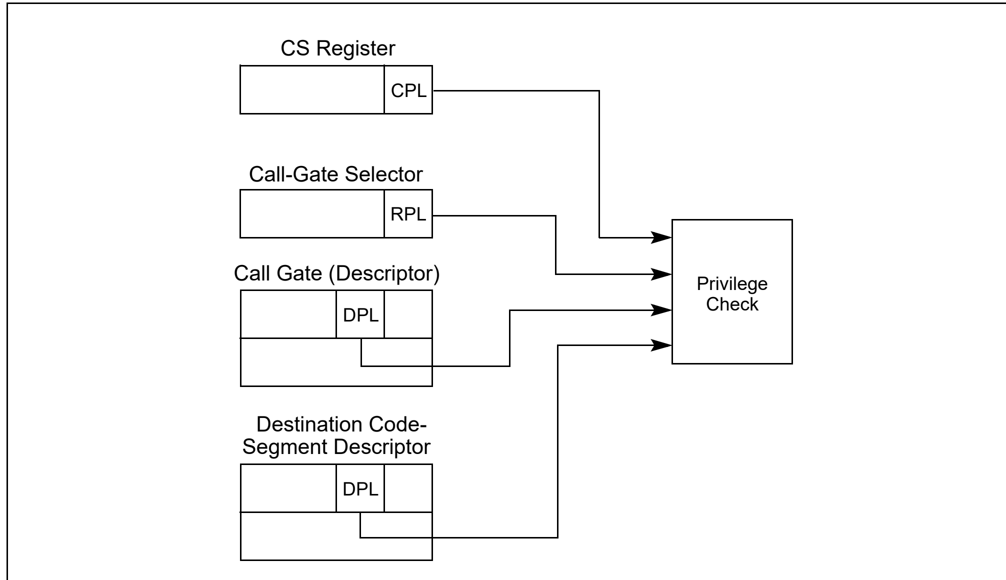


Figure 5-11. Privilege Check for Control Transfer with Call Gate

The privilege checking rules are different depending on whether the control transfer was initiated with a CALL or a JMP instruction, as shown in Table 5-1.

Table 5-1. Privilege Check Rules for Call Gates

Instruction	Privilege Check Rules
CALL	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL ≤ CPL
JMP	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL = CPL

The DPL field of the call-gate descriptor specifies the numerically highest privilege level from which a calling procedure can access the call gate; that is, to access a call gate, the CPL of a calling procedure must be equal to or less than the DPL of the call gate. For example, in Figure 5-15, call gate A has a DPL of 3. So calling procedures at all CPLs (0 through 3) can access this call gate, which includes calling procedures in code segments A, B, and C. Call gate B has a DPL of 2, so only calling procedures at a CPL or 0, 1, or 2 can access call gate B, which includes calling procedures in code segments B and C. The dotted line shows that a calling procedure in code segment A cannot access call gate B.

The RPL of the segment selector to a call gate must satisfy the same test as the CPL of the calling procedure; that is, the RPL must be less than or equal to the DPL of the call gate. In the example in Figure 5-15, a calling procedure in code segment C can access call gate B using gate selector B2 or B1, but it could not use gate selector B3 to access call gate B.

If the privilege checks between the calling procedure and call gate are successful, the processor then checks the DPL of the code-segment descriptor against the CPL of the calling procedure. Here, the privilege check rules vary between CALL and JMP instructions. Only CALL instructions can use call gates to transfer program control to more privileged (numerically lower privilege level) nonconforming code segments; that is, to nonconforming code segments with a DPL less than the CPL. A JMP instruction can use a call gate only to transfer program control to a nonconforming code segment with a DPL equal to the CPL. CALL and JMP instruction can both transfer program control to a more privileged conforming code segment; that is, to a conforming code segment with a DPL less than or equal to the CPL.

If a call is made to a more privileged (numerically lower privilege level) nonconforming destination code segment, the CPL is lowered to the DPL of the destination code segment and a stack switch occurs (see Section 5.8.5, "Stack Switching"). If a call or jump is made to a more privileged conforming destination code segment, the CPL is not changed and no stack switch occurs.

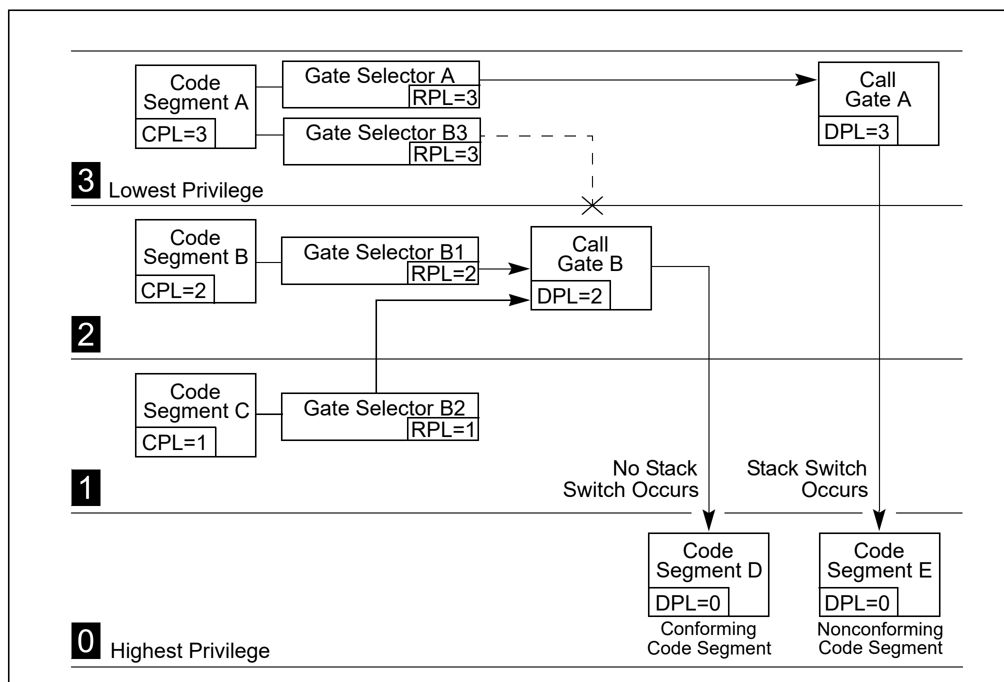


Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels

Call gates allow a single code segment to have procedures that can be accessed at different privilege levels. For example, an operating system located in a code segment may have some services which are intended to be used by both the operating system and application software (such as procedures for handling character I/O). Call gates for these procedures can be set up that allow access at all privilege levels (0 through 3). More privileged call gates (with DPLs of 0 or 1) can then be set up for other operating system services that are intended to be used only by the operating system (such as procedures that initialize device drivers).

5.8.5 Stack Switching

Whenever a call gate is used to transfer program control to a more privileged nonconforming code segment (that is, when the DPL of the nonconforming destination code segment is less than the CPL), the processor automatically switches to the stack for the destination code segment's privilege level. This stack switching is carried out to prevent more privileged procedures from crashing due to insufficient stack space. It also prevents less privileged procedures from interfering (by accident or intent) with more privileged procedures through a shared stack.

Each task must define up to 4 stacks: one for applications code (running at privilege level 3) and one for each of the privilege levels 2, 1, and 0 that are used. (If only two privilege levels are used [3 and 0], then only two stacks must be defined.) Each of these stacks is located in a separate segment and is identified with a segment selector and an offset into the stack segment (a stack pointer).

The segment selector and stack pointer for the privilege level 3 stack is located in the SS and ESP registers, respectively, when privilege-level-3 code is being executed and is automatically stored on the called procedure's stack when a stack switch occurs.

Pointers to the privilege level 0, 1, and 2 stacks are stored in the TSS for the currently running task (see Figure 7-2). Each of these pointers consists of a segment selector and a stack pointer (loaded into the ESP register). These initial pointers are strictly read-only values. The processor does not change them while the task is running. They are used only to create new stacks when calls are made to more privileged levels (numerically lower

privilege levels). These stacks are disposed of when a return is made from the called procedure. The next time the procedure is called, a new stack is created using the initial stack pointer. (The TSS does not specify a stack for privilege level 3 because the processor does not allow a transfer of program control from a procedure running at a CPL of 0, 1, or 2 to a procedure running at a CPL of 3, except on a return.)

The operating system is responsible for creating stacks and stack-segment descriptors for all the privilege levels to be used and for loading initial pointers for these stacks into the TSS. Each stack must be read/write accessible (as specified in the type field of its segment descriptor) and must contain enough space (as specified in the limit field) to hold the following items:

- The contents of the SS, ESP, CS, and EIP registers for the calling procedure.
- The parameters and temporary variables required by the called procedure.
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

The stack will need to require enough space to contain many frames of these items, because procedures often call other procedures, and an operating system may support nesting of multiple interrupts. Each stack should be large enough to allow for the worst case nesting scenario at its privilege level.

(If the operating system does not use the processor's multitasking mechanism, it still must create at least one TSS for this stack-related purpose.)

When a procedure call through a call gate results in a change in privilege level, the processor performs the following steps to switch stacks and begin execution of the called procedure at a new privilege level:

1. Uses the DPL of the destination code segment (the new CPL) to select a pointer to the new stack (segment selector and stack pointer) from the TSS.
2. Reads the segment selector and stack pointer for the stack to be switched to from the current TSS. Any limit violations detected while reading the stack-segment selector, stack pointer, or stack-segment descriptor cause an invalid TSS (#TS) exception to be generated.
3. Checks the stack-segment descriptor for the proper privileges and type and generates an invalid TSS (#TS) exception if violations are detected.
4. Temporarily saves the current values of the SS and ESP registers.
5. Loads the segment selector and stack pointer for the new stack in the SS and ESP registers.
6. Pushes the temporarily saved values for the SS and ESP registers (for the calling procedure) onto the new stack (see Figure 5-13).
7. Copies the number of parameter specified in the parameter count field of the call gate from the calling procedure's stack to the new stack. If the count is 0, no parameters are copied.
8. Pushes the return instruction pointer (the current contents of the CS and EIP registers) onto the new stack.
9. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively, and begins execution of the called procedure.

See the description of the CALL instruction in Chapter 3, *Instruction Set Reference*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far call through a call gate.

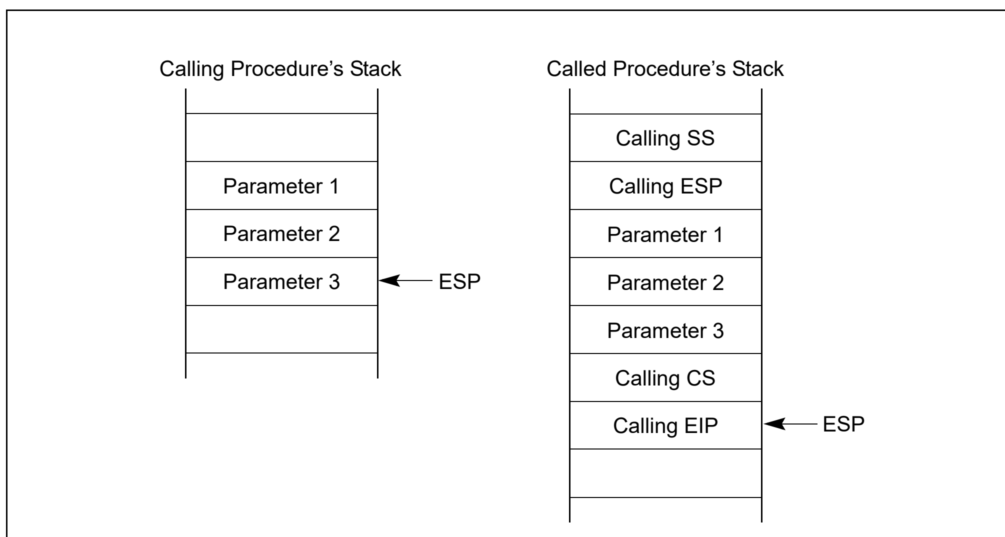


Figure 5-13. Stack Switching During an Interprivilege-Level Call

The parameter count field in a call gate specifies the number of data items (up to 31) that the processor should copy from the calling procedure's stack to the stack of the called procedure. If more than 31 data items need to be passed to the called procedure, one of the parameters can be a pointer to a data structure, or the saved contents of the SS and ESP registers may be used to access parameters in the old stack space. The size of the data items passed to the called procedure depends on the call gate size, as described in Section 5.8.3, "Call Gates."

5.8.5.1 Stack Switching in 64-bit Mode

Although protection-check rules for call gates are unchanged from 32-bit mode, stack-switch changes in 64-bit mode are different.

When stacks are switched as part of a 64-bit mode privilege-level change through a call gate, a new SS (stack segment) descriptor is not loaded; 64-bit mode only loads an inner-level RSP from the TSS. The new SS is forced to NULL and the SS selector's RPL field is forced to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INTn, interrupts and exceptions). The old SS and RSP are saved on the new stack.

On a subsequent far RET, the old SS is popped from the stack and loaded into the SS register. See Table 5-2.

Table 5-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change

32-bit Mode			IA-32e mode		
Old SS Selector	+12			+24	Old SS Selector
Old ESP	+8			+16	Old RSP
CS Selector	+4			+8	Old CS Selector
EIP	0	**ESP**	**RSP**	0	RIP
< 4 Bytes >					< 8 Bytes >

In 64-bit mode, stack operations resulting from a privilege-level-changing far call or far return are eight-bytes wide and change the RSP by eight. The mode does not support the automatic parameter-copy feature found in 32-bit mode. The call-gate count field is ignored. Software can access the old stack, if necessary, by referencing the old stack-segment selector and stack pointer saved on the new process stack.

In 64-bit mode, far RET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL ≠ 3, IRET allows SS to be loaded with a NULL selector. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent far RET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

5.8.6 Returning from a Called Procedure

The RET instruction can be used to perform a near return, a far return at the same privilege level, and a far return to a different privilege level. This instruction is intended to execute returns from procedures that were called with a CALL instruction. It does not support returns from a JMP instruction, because the JMP instruction does not save a return instruction pointer on the stack.

A near return only transfers program control within the current code segment; therefore, the processor performs only a limit check. When the processor pops the return instruction pointer from the stack into the EIP register, it checks that the pointer does not exceed the limit of the current code segment.

On a far return at the same privilege level, the processor pops both a segment selector for the code segment being returned to and a return instruction pointer from the stack. Under normal conditions, these pointers should be valid, because they were pushed on the stack by the CALL instruction. However, the processor performs privilege checks to detect situations where the current procedure might have altered the pointer or failed to maintain the stack properly.

A far return that requires a privilege-level change is only allowed when returning to a less privileged level (that is, the DPL of the return code segment is numerically greater than the CPL). The processor uses the RPL field from the CS register value saved for the calling procedure (see Figure 5-13) to determine if a return to a numerically higher privilege level is required. If the RPL is numerically greater (less privileged) than the CPL, a return across privilege levels occurs.

The processor performs the following steps when performing a far return to a calling procedure (see Figures 6-2 and 6-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of the stack contents prior to and after a return):

1. Checks the RPL field of the saved CS register value to determine if a privilege level change is required on the return.
2. Loads the CS and EIP registers with the values on the called procedure's stack. (Type and privilege level checks are performed on the code-segment descriptor and RPL of the code- segment selector.)
3. (If the RET instruction includes a parameter count operand and the return requires a privilege level change.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value (after popping the CS and EIP values), to step past the parameters on the called procedure's stack. The resulting value in the ESP register points to the saved SS and ESP values for the calling procedure's stack. (Note that the byte count in the RET instruction must be chosen to match the parameter count in the call gate that the calling procedure referenced when it made the original call multiplied by the size of the parameters.)
4. (If the return requires a privilege level change.) Loads the SS and ESP registers with the saved SS and ESP values and switches back to the calling procedure's stack. The SS and ESP values for the called procedure's stack are discarded. Any limit violations detected while loading the stack-segment selector or stack pointer cause a general-protection exception (#GP) to be generated. The new stack-segment descriptor is also checked for type and privilege violations.
5. (If the RET instruction includes a parameter count operand.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value, to step past the parameters on the calling procedure's stack. The resulting ESP value is not checked against the limit of the stack segment. If the ESP value is beyond the limit, that fact is not recognized until the next stack operation.
6. (If the return requires a privilege level change.) Checks the contents of the DS, ES, FS, and GS segment registers. If any of these registers refer to segments whose DPL is less than the new CPL (excluding conforming code segments), the segment register is loaded with a null segment selector.

See the description of the RET instruction in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far return.

5.8.7 Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processors for the purpose of providing a fast (low overhead) mechanism for calling operating system or executive procedures.

SYSENTER is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSEXIT is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code. SYSENTER can be executed from privilege levels 3, 2, 1, or 0; SYSEXIT can only be executed from privilege level 0.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. This is because SYSENTER does not save any state information for use by SYSEXIT on a return.

The target instruction and stack pointer for these instructions are not specified through instruction operands. Instead, they are specified through parameters entered in MSRs and general-purpose registers.

For SYSENTER, target fields are generated using the following sources:

- **Target code segment** — Reads this from IA32_SYSENTER_CS.
- **Target instruction** — Reads this from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from the IA32_SYSENTER_ESP.

For SYSEXIT, target fields are generated using the following sources:

- **Target code segment** — Computed by adding 16 to the value in the IA32_SYSENTER_CS.
- **Target instruction** — Reads this from EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from ECX.

The SYSENTER and SYSEXIT instructions preform "fast" calls and returns because they force the processor into a predefined privilege level 0 state when SYSENTER is executed and into a predefined privilege level 3 state when SYSEXIT is executed. By forcing predefined and consistent processor states, the number of privilege checks ordinarily required to perform a far call to another privilege levels are greatly reduced. Also, by predefining the target context state in MSRs and general-purpose registers eliminates all memory accesses except when fetching the target code.

Any additional state that needs to be saved to allow a return to the calling procedure must be saved explicitly by the calling procedure or be predefined through programming conventions.

5.8.7.1 SYSENTER and SYSEXIT Instructions in IA-32e Mode

For Intel 64 processors, the SYSENTER and SYSEXIT instructions are enhanced to allow fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. IA32_SYSENTER_EIP MSR and IA32_SYSENTER_ESP MSR are expanded to hold 64-bit addresses. If IA-32e mode is inactive, only the lower 32-bit addresses stored in these MSRs are used. The WRMSR instruction ensures that the addresses stored in these MSRs are canonical. Note that, in 64-bit mode, IA32_SYSENTER_CS must not contain a NULL selector.

When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32_SYSENTER_CS.
- **New CS attributes** — CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32_SYSENTER_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32_SYSENTER_ESP.
- **New SS attributes** — SS base = 0, SS limit = FFFFFFFFH.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 40 to the value of IA32_SYSENTER_CS.

- **Stack pointer** — Update RSP using 64-bit canonical address in RCX.

When SYSEXIT transfers control to compatibility mode user code when the operand size attribute is 32 bits, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 16 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 0 (go to compatibility mode).
- **Target instruction** — Fetch the target instruction from 32-bit address in EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Update ESP from 32-bit address in ECX.

5.8.8 Fast System Calls in 64-Bit Mode

The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used). The instructions, along with SYSENTER and SYSEXIT, are suited for IA-32e mode operation. SYSCALL and SYSRET, however, are not supported in compatibility mode (or in protected mode). Use CPUID to check if SYSCALL and SYSRET are available (CPUID.80000001H.EDX[bit 11] = 1).

SYSCALL is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSRET is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code.

Stack pointers for SYSCALL/SYSRET are not specified through model specific registers. The clearing of bits in RFLAGS is programmable rather than fixed. SYSCALL/SYSRET save and restore the RFLAGS register.

For SYSCALL, the processor saves RFLAGS into R11 and the RIP of the next instruction into RCX; it then gets the privilege-level 0 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[47:32].
- **Target instruction pointer** — Reads a 64-bit address from IA32_LSTAR. (The WRMSR instruction ensures that the value of the IA32_LSTAR MSR is canonical.)
- **Stack segment** — Computed by adding 8 to the value in IA32_STAR[47:32].
- **Flags** — The processor sets RFLAGS to the logical-AND of its current value with the complement of the value in the IA32_FMASK MSR.

When SYSRET transfers control to 64-bit mode user code using REX.W, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48] + 16.
- **Target instruction pointer** — Copies the value in RCX into RIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

When SYSRET transfers control to 32-bit mode user code using a 32-bit operand size, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48].
- **Target instruction pointer** — Copies the value in ECX into EIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

It is the responsibility of the OS to ensure the descriptors in the GDT/LDT correspond to the selectors loaded by SYSCALL/SYSRET (consistent with the base, limit, and attribute values forced by the instructions).

See Figure 5-14 for the layout of IA32_STAR, IA32_LSTAR and IA32_FMASK.

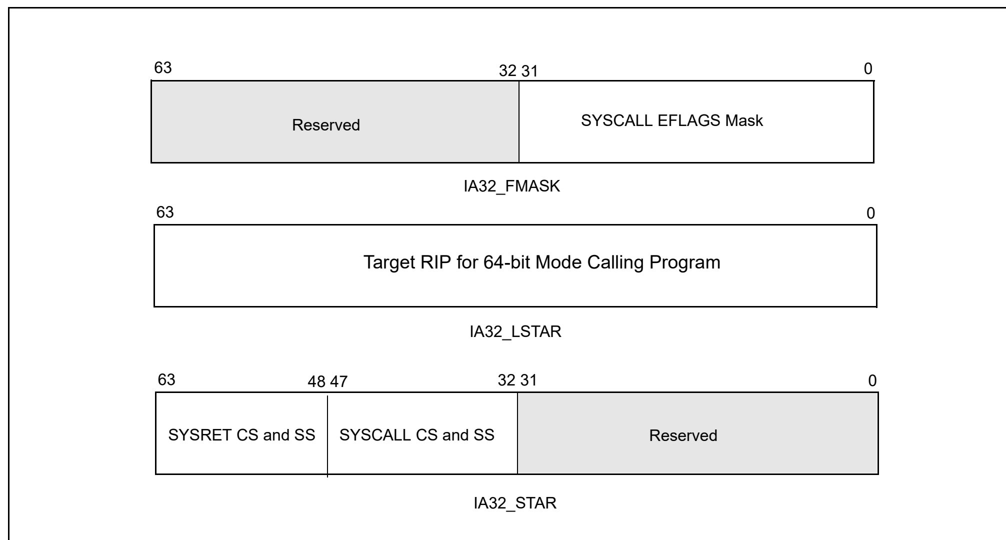


Figure 5-14. MSRs Used by SYSCALL and SYSRET

The SYSCALL instruction does not save the stack pointer, and the SYSRET instruction does not restore it. It is likely that the OS system-call handler will change the stack pointer from the user stack to the OS stack. If so, it is the responsibility of software first to save the user stack pointer. This might be done by user code, prior to executing SYSCALL, or by the OS system-call handler after SYSCALL.

Because the SYSRET instruction does not modify the stack pointer, it is necessary for software to switch back to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table”).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

5.9 PRIVILEGED INSTRUCTIONS

Some of the system instructions (called “privileged instructions”) are protected from use by application programs. The privileged instructions control system functions (such as the loading of system registers). They can be executed only when the CPL is 0 (most privileged). If one of these instructions is executed when the CPL is not 0, a general-protection exception (#GP) is generated. The following system instructions are privileged instructions:

- LGDT — Load GDT register.
- LLDT — Load LDT register.

- LTR — Load task register.
- LIDT — Load IDT register.
- MOV (control registers) — Load and store control registers.
- LMSW — Load machine status word.
- CLTS — Clear task-switched flag in register CR0.
- MOV (debug registers) — Load and store debug registers.
- INVD — Invalidate cache, without writeback.
- WBINVD — Invalidate cache, with writeback.
- INVLPG —Invalidate TLB entry.
- HLT— Halt processor.
- RDMSR — Read Model-Specific Registers.
- WRMSR —Write Model-Specific Registers.
- RDPMC — Read Performance-Monitoring Counter.
- RDTSC — Read Time-Stamp Counter.

Some of the privileged instructions are available only in the more recent families of Intel 64 and IA-32 processors (see Section 22.13, "New Instructions In the Pentium and Later IA-32 Processors").

The PCE and TSD flags in register CR4 (bits 4 and 2, respectively) enable the RDPMC and RDTSC instructions, respectively, to be executed at any CPL.

5.10 POINTER VALIDATION

When operating in protected mode, the processor validates all pointers to enforce protection between segments and maintain isolation between privilege levels. Pointer validation consists of the following checks:

1. Checking access rights to determine if the segment type is compatible with its use.
2. Checking read/write rights.
3. Checking if the pointer offset exceeds the segment limit.
4. Checking if the supplier of the pointer is allowed to access the segment.
5. Checking the offset alignment.

The processor automatically performs first, second, and third checks during instruction execution. Software must explicitly request the fourth check by issuing an ARPL instruction. The fifth check (offset alignment) is performed automatically at privilege level 3 if alignment checking is turned on. Offset alignment does not affect isolation of privilege levels.

5.10.1 Checking Access Rights (LAR Instruction)

When the processor accesses a segment using a far pointer, it performs an access rights check on the segment descriptor pointed to by the far pointer. This check is performed to determine if type and privilege level (DPL) of the segment descriptor are compatible with the operation to be performed. For example, when making a far call in protected mode, the segment-descriptor type must be for a conforming or nonconforming code segment, a call gate, a task gate, or a TSS. Then, if the call is to a nonconforming code segment, the DPL of the code segment must be equal to the CPL, and the RPL of the code segment's segment selector must be less than or equal to the DPL. If type or privilege level are found to be incompatible, the appropriate exception is generated.

To prevent type incompatibility exceptions from being generated, software can check the access rights of a segment descriptor using the LAR (load access rights) instruction. The LAR instruction specifies the segment selector for the segment descriptor whose access rights are to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.

2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, call gate, task gate, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the second doubleword of the segment descriptor into the destination register (masked by the value 00FXFF00H, where X indicates that the corresponding 4 bits are undefined) and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LAR instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can preform additional checks on the access rights information.

5.10.2 Checking Read/Write Rights (VERR and VERW Instructions)

When the processor accesses any code or data segment it checks the read/write privileges assigned to the segment to verify that the intended read or write operation is allowed. Software can check read/write rights using the VERR (verify for reading) and VERW (verify for writing) instructions. Both these instructions specify the segment selector for the segment being checked. The instructions then perform the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code or data-segment descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. Checks that the segment is readable (for the VERR instruction) or writable (for the VERW) instruction.

The VERR instruction sets the ZF flag in the EFLAGS register if the segment is visible at the CPL and readable; the VERW sets the ZF flag if the segment is visible and writable. (Code segments are never writable.) The ZF flag is cleared if any of these checks fail.

5.10.3 Checking That the Pointer Offset Is Within Limits (LSL Instruction)

When the processor accesses any segment it performs a limit check to insure that the offset is within the limit of the segment. Software can perform this limit check using the LSL (load segment limit) instruction. Like the LAR instruction, the LSL instruction specifies the segment selector for the segment descriptor whose limit is to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the unscrambled limit (the limit scaled according to the setting of the G flag in the segment descriptor) into the destination register and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LSL instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

5.10.4 Checking Caller Access Privileges (ARPL Instruction)

The requestor's privilege level (RPL) field of a segment selector is intended to carry the privilege level of a calling procedure (the calling procedure's CPL) to a called procedure. The called procedure then uses the RPL to determine if access to a segment is allowed. The RPL is said to "weaken" the privilege level of the called procedure to that of the RPL.

Operating-system procedures typically use the RPL to prevent less privileged application programs from accessing data located in more privileged segments. When an operating-system procedure (the called procedure) receives a segment selector from an application program (the calling procedure), it sets the segment selector's RPL to the privilege level of the calling procedure. Then, when the operating system uses the segment selector to access its associated segment, the processor performs privilege checks using the calling procedure's privilege level (stored in the RPL) rather than the numerically lower privilege level (the CPL) of the operating-system procedure. The RPL thus insures that the operating system does not access a segment on behalf of an application program unless that program itself has access to the segment.

Figure 5-15 shows an example of how the processor uses the RPL field. In this example, an application program (located in code segment A) possesses a segment selector (segment selector D1) that points to a privileged data structure (that is, a data structure located in a data segment D at privilege level 0).

The application program cannot access data segment D, because it does not have sufficient privilege, but the operating system (located in code segment C) can. So, in an attempt to access data segment D, the application program executes a call to the operating system and passes segment selector D1 to the operating system as a parameter on the stack. Before passing the segment selector, the (well behaved) application program sets the RPL of the segment selector to its current privilege level (which in this example is 3). If the operating system attempts to access data segment D using segment selector D1, the processor compares the CPL (which is now 0 following the call), the RPL of segment selector D1, and the DPL of data segment D (which is 0). Since the RPL is greater than the DPL, access to data segment D is denied. The processor's protection mechanism thus protects data segment D from access by the operating system, because application program's privilege level (represented by the RPL of segment selector B) is greater than the DPL of data segment D.

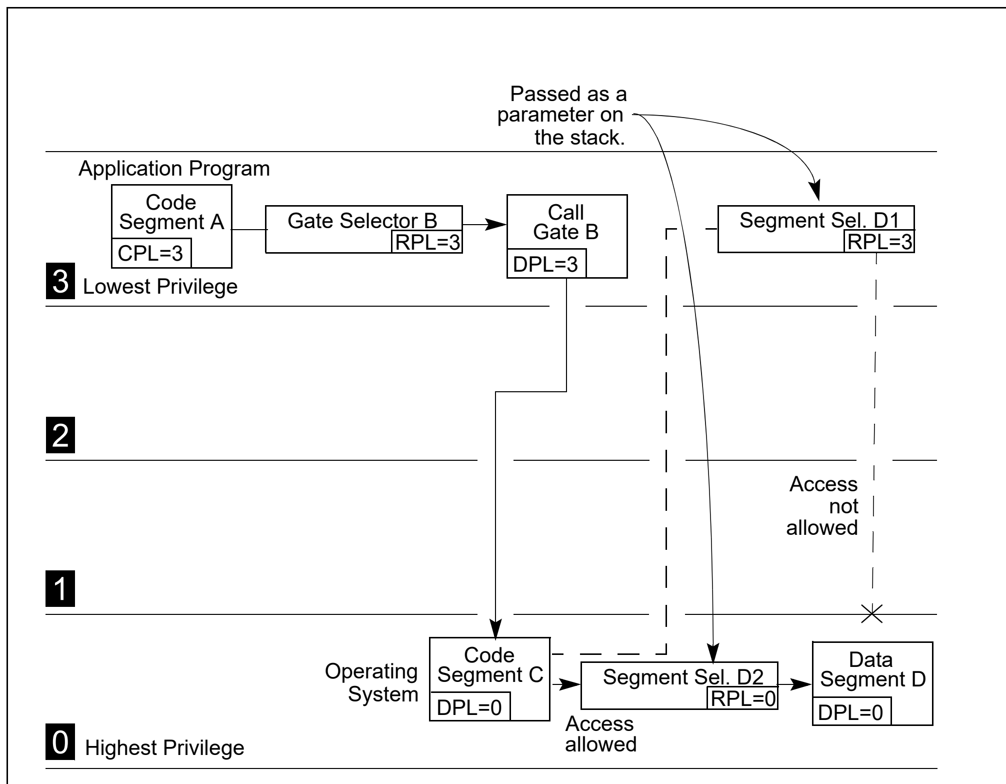


Figure 5-15. Use of RPL to Weaken Privilege Level of Called Procedure

Now assume that instead of setting the RPL of the segment selector to 3, the application program sets the RPL to 0 (segment selector D2). The operating system can now access data segment D, because its CPL and the RPL of segment selector D2 are both equal to the DPL of data segment D.

Because the application program is able to change the RPL of a segment selector to any value, it can potentially use a procedure operating at a numerically lower privilege level to access a protected data structure. This ability to lower the RPL of a segment selector breaches the processor's protection mechanism.

Because a called procedure cannot rely on the calling procedure to set the RPL correctly, operating-system procedures (executing at numerically lower privilege-levels) that receive segment selectors from numerically higher privilege-level procedures need to test the RPL of the segment selector to determine if it is at the appropriate level. The ARPL (adjust requested privilege level) instruction is provided for this purpose. This instruction adjusts the RPL of one segment selector to match that of another segment selector.

The example in Figure 5-15 demonstrates how the ARPL instruction is intended to be used. When the operating-system receives segment selector D2 from the application program, it uses the ARPL instruction to compare the RPL of the segment selector with the privilege level of the application program (represented by the code-segment selector pushed onto the stack). If the RPL is less than application program's privilege level, the ARPL instruction changes the RPL of the segment selector to match the privilege level of the application program (segment selector D1). Using this instruction thus prevents a procedure running at a numerically higher privilege level from accessing numerically lower privilege-level (more privileged) segments by lowering the RPL of a segment selector.

Note that the privilege level of the application program can be determined by reading the RPL field of the segment selector for the application-program's code segment. This segment selector is stored on the stack as part of the call to the operating system. The operating system can copy the segment selector from the stack into a register for use as an operand for the ARPL instruction.

5.10.5 Checking Alignment

When the CPL is 3, alignment of memory references can be checked by setting the AM flag in the CR0 register and the AC flag in the EFLAGS register. Unaligned memory references generate alignment exceptions (#AC). The processor does not generate alignment exceptions when operating at privilege level 0, 1, or 2. See Table 6-7 for a description of the alignment requirements when alignment checking is enabled.

5.11 PAGE-LEVEL PROTECTION

Page-level protection can be used alone or applied to segments. When page-level protection is used with the flat memory model, it allows supervisor code and data (the operating system or executive) to be protected from user code and data (application programs). It also allows pages containing code to be write protected. When the segment- and page-level protection are combined, page-level read/write protection allows more protection granularity within segments.

With page-level protection (as with segment-level protection) each memory reference is checked to verify that protection checks are satisfied. All checks are made before the memory cycle is started, and any violation prevents the cycle from starting and results in a page-fault exception being generated. Because checks are performed in parallel with address translation, there is no performance penalty.

The processor performs two page-level protection checks:

- Restriction of addressable domain (supervisor and user modes).
- Page type (read only or read/write).

Violations of either of these checks results in a page-fault exception being generated. See Chapter 6, "Interrupt 14—Page-Fault Exception (#PF)," for an explanation of the page-fault exception mechanism. This chapter describes the protection violations which lead to page-fault exceptions.

5.11.1 Page-Protection Flags

Protection information for pages is contained in two flags in a paging-structure entry (see Chapter 4): the read/write flag (bit 1) and the user/supervisor flag (bit 2). The protection checks use the flags in all paging structures.

5.11.2 Restricting Addressable Domain

The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is in supervisor mode, it can access all pages; when in user mode, it can access only user-level pages. (Note that the WP flag in control register CR0 modifies the supervisor permissions, as described in Section 5.11.3, "Page Type.")

Note that to use the page-level protection mechanism, code and data segments must be set up for at least two segment-based privilege levels: level 0 for supervisor code and data segments and level 3 for user code and data segments. (In this model, the stacks are placed in the data segments.) To minimize the use of segments, a flat memory model can be used (see Section 3.2.1, "Basic Flat Model").

Here, the user and supervisor code and data segments all begin at address zero in the linear address space and overlay each other. With this arrangement, operating-system code (running at the supervisor level) and application code (running at the user level) can execute as if there are no segments. Protection between operating-system and application code and data is provided by the processor's page-level protection mechanism.

5.11.3 Page Type

The page-level protection mechanism recognizes two page types:

- Read-only access (R/W flag is 0).
- Read/write access (R/W flag is 1).

When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the processor is in user mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode. A page-fault exception is generated on any attempt to violate the protection rules.

Starting with the P6 family, Intel processors allow user-mode pages to be write-protected against supervisor-mode access. Setting CR0.WP = 1 enables supervisor-mode sensitivity to write protected pages. If CR0.WP = 1, read-only pages are not writable from any privilege level. This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. An alternative copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page. By using the WP flag and marking the shared pages as read-only, the supervisor can detect an attempt to write to a page, and can copy the page at that time.

5.11.4 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 5-3 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.

5.11.5 Overrides to Page Protection

The following types of memory accesses are checked as if they are privilege-level 0 accesses, regardless of the CPL at which the processor is currently operating:

- Access to segment descriptors in the GDT, LDT, or IDT.
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

5.12 COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Page-level protections cannot be used to override segment-level protection. For example, a code segment is by definition not writable. If a code segment is paged, setting the R/W flag for the pages to read-write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

Page-level protection can be used to enhance segment-level protection. For example, if a large read-write data segment is paged, the page-protection mechanism can be used to write-protect individual pages.

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

5.13 PAGE-LEVEL PROTECTION AND EXECUTE-DISABLE BIT

In addition to page-level protection offered by the U/S and R/W flags, paging structures used with PAE paging and 4-level paging[1] (see Chapter 4) provide the execute-disable bit. This bit offers additional protection for data pages.

An Intel 64 or IA-32 processor with the execute-disable bit capability can prevent data pages from being used by malicious software to execute code. This capability is provided in:

- 32-bit protected mode with PAE enabled.
- IA-32e mode.

While the execute-disable bit capability does not introduce new instructions, it does require operating systems to use a PAE-enabled environment and establish a page-granular protection policy for memory pages.

If the execute-disable bit of a memory page is set, that page can be used only as data. An attempt to execute code from a memory page with the execute-disable bit set causes a page-fault exception.

The execute-disable capability is supported only with PAE paging and 4-level paging. It is not supported with 32-bit paging. Existing page-level protection mechanisms (see Section 5.11, “Page-Level Protection”) continue to apply to memory pages independent of the execute-disable setting.

5.13.1 Detecting and Enabling the Execute-Disable Capability

Software can detect the presence of the execute-disable capability using the CPUID instruction. CPUID.80000001H:EDX.NX [bit 20] = 1 indicates the capability is available.

If the capability is available, software can enable it by setting IA32_EFER.NXE[bit 11] to 1. IA32_EFER is available if CPUID.80000001H.EDX[bit 20 or 29] = 1.

If the execute-disable capability is not available, a write to set IA32_EFER.NXE produces a #GP exception. See Table 5-4.

Table 5-4. Extended Feature Enable MSR (IA32_EFER)

63:12	11	10	9	8	7:1	0
Reserved	Execute-disable bit enable (NXE)	IA-32e mode active (LMA)	Reserved	IA-32e mode enable (LME)	Reserved	SysCall enable (SCE)

5.13.2 Execute-Disable Page Protection

The execute-disable bit in the paging structures enhances page protection for data pages. Instructions cannot be fetched from a memory page if IA32_EFER.NXE =1 and the execute-disable bit is set in any of the paging-structure entries used to map the page. Table 5-5 lists the valid usage of a page in relation to the value of execute-disable bit (bit 63) of the corresponding entry in each level of the paging structures. Execute-disable protection can be activated using the execute-disable bit at any level of the paging structure, irrespective of the corresponding entry in other levels. When execute-disable protection is not activated, the page can be used as code or data.

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

Table 5-5. IA-32e Mode Page Level Protection Matrix with Execute-Disable Bit Capability

Execute Disable Bit Value (Bit 63)				Valid Usage
PML4	PDP	PDE	PTE	
Bit 63 = 1	*	*	*	Data
*	Bit 63 = 1	*	*	Data
*	*	Bit 63 = 1	*	Data
*	*	*	Bit 63 = 1	Data
Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Data/Code

NOTES:

* Value not checked.

In legacy PAE-enabled mode, Table 5-6 and Table 5-7 show the effect of setting the execute-disable bit for code and data pages.

Table 5-6. Legacy PAE-Enabled 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability

Execute Disable Bit Value (Bit 63)		Valid Usage
PDE	PTE	
Bit 63 = 1	*	Data
*	Bit 63 = 1	Data
Bit 63 = 0	Bit 63 = 0	Data/Code

NOTE:

* Value not checked.

Table 5-7. Legacy PAE-Enabled 2-MByte Page Level Protection with Execute-Disable Bit Capability

Execute Disable Bit Value (Bit 63)	Valid Usage
PDE	
Bit 63 = 1	Data
Bit 63 = 0	Data/Code

5.13.3 Reserved Bit Checking

The processor enforces reserved bit checking in paging data structure entries. The bits being checked varies with paging mode and may vary with the size of physical address space.

Table 5-8 shows the reserved bits that are checked when the execute disable bit capability is enabled (CR4.PAE = 1 and IA32_EFER.NXE = 1). Table 5-8 and Table 5-9 show the following paging modes:

- Non-PAE 4-KByte paging: 4-KByte-page only paging (CR4.PAE = 0, CR4.PSE = 0).
- PSE36: 4-KByte and 4-MByte pages (CR4.PAE = 0, CR4.PSE = 1).
- PAE: 4-KByte and 2-MByte pages (CR4.PAE = 1, CR4.PSE = X).

The reserved bit checking depends on the physical address size supported by the implementation, which is reported in CPUID.80000008H. See the table note.

Table 5-8. IA-32e Mode Page Level Protection Matrix with Execute-Disable Bit Capability Enabled

Mode	Paging Mode	Check Bits
32-bit	4-KByte paging (non-PAE)	No reserved bits checked
	PSE36 - PDE, 4-MByte page	Bit [21]
	PSE36 - PDE, 4-KByte page	No reserved bits checked
	PSE36 - PTE	No reserved bits checked
	PAE - PDP table entry	Bits [63:MAXPHYADDR] & [8:5] & [2:1] *
	PAE - PDE, 2-MByte page	Bits [62:MAXPHYADDR] & [20:13] *
	PAE - PDE, 4-KByte page	Bits [62:MAXPHYADDR] *
	PAE - PTE	Bits [62:MAXPHYADDR] *
64-bit	PML4E	Bits [51:MAXPHYADDR] *
	PDPTE	Bits [51:MAXPHYADDR] *
	PDE, 2-MByte page	Bits [51:MAXPHYADDR] & [20:13] *
	PDE, 4-KByte page	Bits [51:MAXPHYADDR] *
	PTE	Bits [51:MAXPHYADDR] *

NOTES:

* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

If execute disable bit capability is not enabled or not available, reserved bit checking in 64-bit mode includes bit 63 and additional bits. This and reserved bit checking for legacy 32-bit paging modes are shown in Table 5-10.

Table 5-9. Reserved Bit Checking WIth Execute-Disable Bit Capability Not Enabled

Mode	Paging Mode	Check Bits
32-bit	KByte paging (non-PAE)	No reserved bits checked
	PSE36 - PDE, 4-MByte page	Bit [21]
	PSE36 - PDE, 4-KByte page	No reserved bits checked
	PSE36 - PTE	No reserved bits checked
	PAE - PDP table entry	Bits [63:MAXPHYADDR] & [8:5] & [2:1]*
	PAE - PDE, 2-MByte page	Bits [63:MAXPHYADDR] & [20:13]*
	PAE - PDE, 4-KByte page	Bits [63:MAXPHYADDR]*
	PAE - PTE	Bits [63:MAXPHYADDR]*
64-bit	PML4E	Bit [63], bits [51:MAXPHYADDR]*
	PDPTE	Bit [63], bits [51:MAXPHYADDR]*
	PDE, 2-MByte page	Bit [63], bits [51:MAXPHYADDR] & [20:13]*
	PDE, 4-KByte page	Bit [63], bits [51:MAXPHYADDR]*
	PTE	Bit [63], bits [51:MAXPHYADDR]*

NOTES:

* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

5.13.4 Exception Handling

When execute disable bit capability is enabled (IA32_EFER.NXE = 1), conditions for a page fault to occur include the same conditions that apply to an Intel 64 or IA-32 processor without execute disable bit capability plus the following new condition: an instruction fetch to a linear address that translates to physical address in a memory page that has the execute-disable bit set.

An Execute Disable Bit page fault can occur at all privilege levels. It can occur on any instruction fetch, including (but not limited to): near branches, far branches, CALL/RET/INT/IRET execution, sequential instruction fetches, and task switches. The execute-disable bit in the page translation mechanism is checked only when:

- IA32_EFER.NXE = 1.
- The instruction translation look-aside buffer (ITLB) is loaded with a page that is not already present in the ITLB.

11. Updates to Chapter 17, Volume 3B

Change bars show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Update to figure 17-23 "IA32_PQR_ASSOC MSR". Update to Section 17.7 "Last Branch, Interrupt and Exception Recording for Intel® Xeon Phi™ Processor 7200/5200/3200".

CHAPTER 17
DEBUG, BRANCH PROFILE, TSC, AND RESOURCE MONITORING FEATURES

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

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.
- Time stamp counter is described in Section 17.16, “Time-Stamp Counter”.
- Features which allow monitoring of shared platform resources such as the L3 cache are described in Section 17.17, “Intel® Resource Director Technology (Intel® RDT) Monitoring Features”.
- Features which enable control over shared platform resources are described in Section 17.18, “Intel® Resource Director Technology (Intel® RDT) Allocation Features”.

17.1 OVERVIEW OF DEBUG SUPPORT FACILITIES

The following processor facilities support debugging and performance monitoring:

- **Debug exception (#DB)** — Transfers program control to a debug procedure or task when a debug event occurs.
- **Breakpoint exception (#BP)** — See breakpoint instruction (INT 3) below.
- **Breakpoint-address registers (DR0 through DR3)** — Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6)** — Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7)** — Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS** — Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register** — Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register** — Generates a debug exception (#DB) after every execution of an instruction.
- **Breakpoint instruction (INT 3)** — Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.
- Execution of the breakpoint instruction.

- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

17.2 DEBUG REGISTERS

Eight debug registers (see Figure 17-1 for 32-bit operation and Figure 17-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

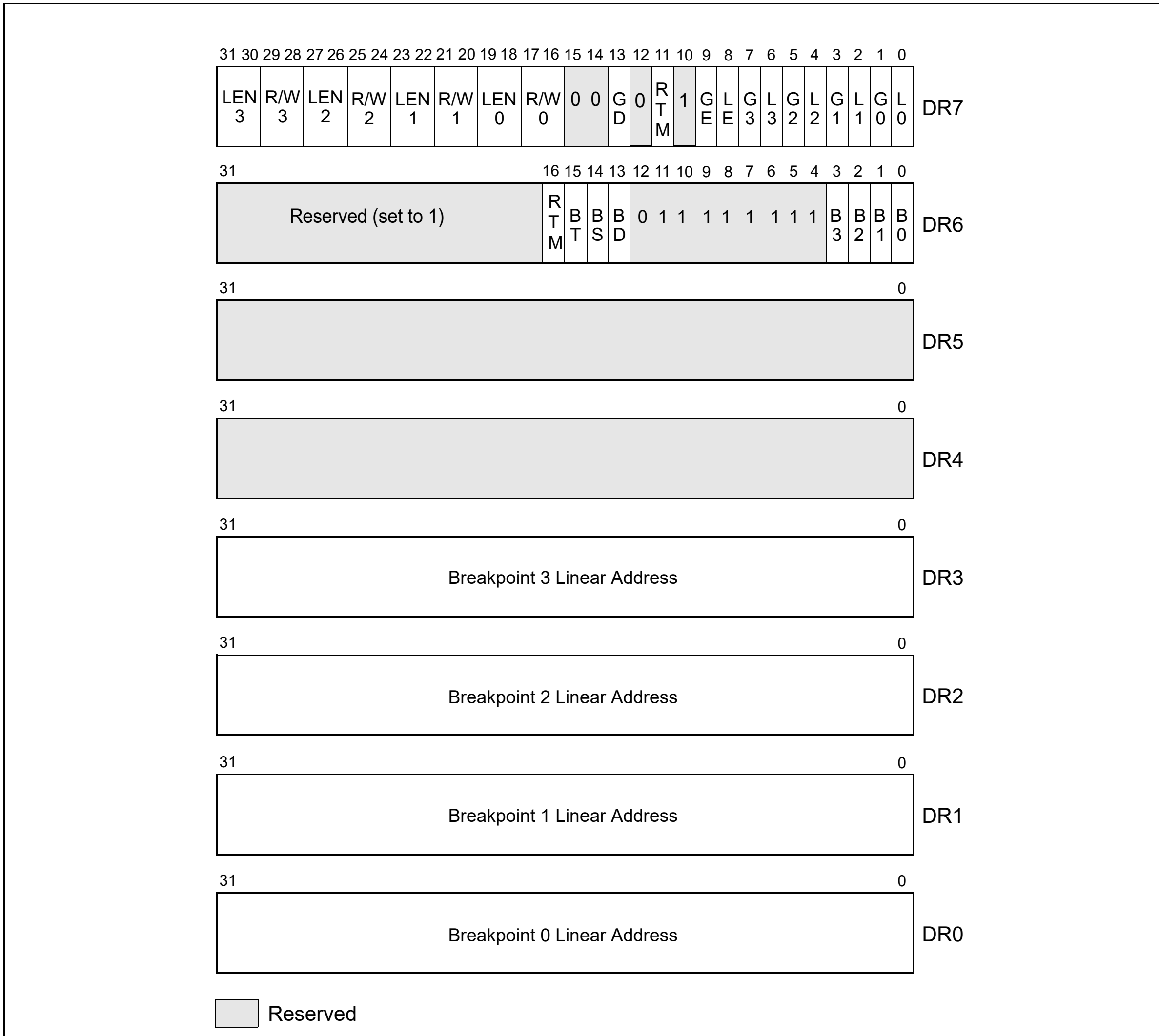


Figure 17-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 though 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 17.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

17.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 17-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

17.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

17.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 17-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3)** — Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LEN*n*, and R/W*n* flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BD (debug register access detected) flag (bit 13)** — Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 17.2.4, "Debug Control Register (DR7)," for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14)** — Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15)** — Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, "Task-State Segment (TSS)," for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when **clear**) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

17.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 17-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)** — Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated L*n* flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7)** — Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated G*n* flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9)** — This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 17.3.3). This advanced debugging is enabled only if IA32_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13)** — Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

  When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

  The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.
- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)** — Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/W*n* fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Break on I/O reads or writes.
  11 — Break on data reads or writes but not instruction fetches.

  When the DE flag is clear, the processor interprets the R/W*n* bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Undefined.
  11 — Break on data reads or writes but not instruction fetches.
- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31)** — Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

  00 — 1-byte length.
  01 — 2-byte length.
  10 — Undefined (or 8 byte length, see note below).
  11 — 4-byte length.

If the corresponding RW*n* field in register DR7 is 00 (instruction execution), then the LEN*n* field should also be 00. The effect of using other lengths is undefined. See Section 17.2.5, “Breakpoint Field Recognition,” below.

NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LEN*n* field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). The Encoding 10B is supported in processors based on Intel® Atom™ microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

17.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the LEN*n* fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The LEN*n* fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DR*n*). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses LEN*n* field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its LEN*n* field. Table 17-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoint addresses must have a length specification of 1 byte (the LEN*n* field is set to 00). Code breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

Table 17-1. Breakpoint Examples

Debug Register Setup			
Debug Register	**R/W*n***	**Breakpoint Address**	**LEN*n***
DR0 DR1 DR2 DR3	R/W0 = 11 (Read/Write) R/W1 = 01 (Write) R/W2 = 11 (Read/Write) R/W3 = 01 (Write)	A0001H A0002H B0002H C0000H	LEN0 = 00 (1 byte) LEN1 = 00 (1 byte) LEN2 = 01) (2 bytes) LEN3 = 11 (4 bytes)

Data Accesses		
Operation	**Address**	**Access Length (In Bytes)**
Data operations that trap - Read or write - Read or write - Write - Write - Read or write - Read or write - Read or write - Write - Write - Write	 A0001H A0001H A0002H A0002H B0001H B0002H B0002H C0000H C0001H C0003H	 1 2 1 2 4 1 2 4 2 1
Data operations that do not trap - Read or write - Read - Read or write - Read or write - Read - Read or write	 A0000H A0002H A0003H B0000H C0000H C0004H	 1 1 4 2 2 4

17.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 17-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Break point conditions for 8-byte memory read/writes are supported in all modes.

17.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

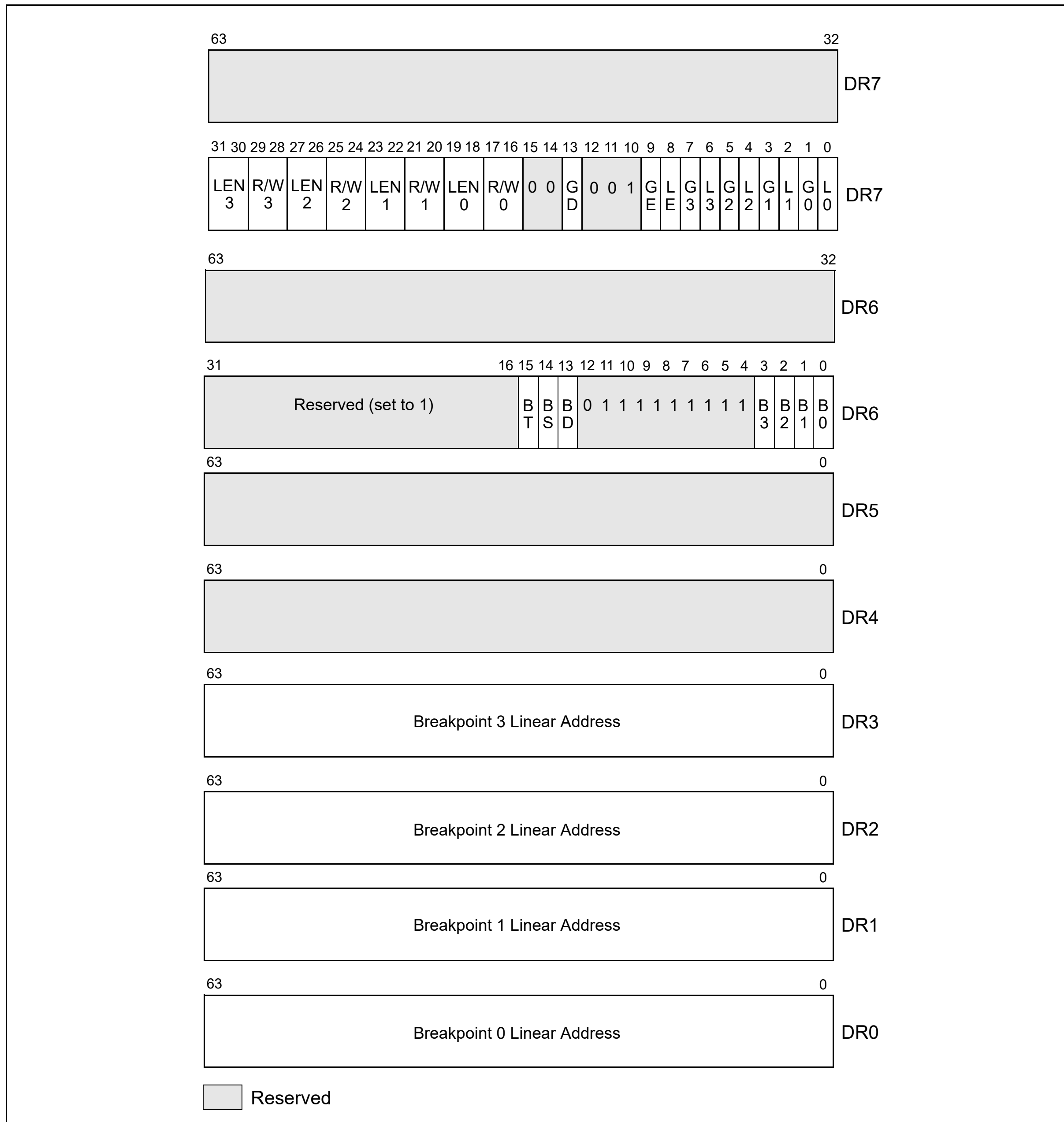


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

17.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 17-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 17.3.1.3, “General-Detect Exception Condition”) result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

See also: Chapter 6, “Interrupt 1—Debug Exception (#DB),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Table 17-2. Debug Exception Conditions

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/W*n* = 0	Fault
Data write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/W*n* = 1	Trap
I/O read or write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/W*n* = 2	Trap
Data read or write (but not instruction fetches), at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/W*n* = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1		Fault
Task switch	BT = 1		Trap

17.3.1.1 Instruction-Breakpoint Exception Condition

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

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see “MOV—Move” and “POP—Pop a Value from the Stack” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, “System Flags and Fields in the EFLAGS Register,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for code breakpoint, CS limit violation and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  — Debug exceptions generated in response to instruction breakpoints

— Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)

— Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)

— Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

17.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

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

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

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

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

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

17.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

17.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n* and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

17.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

17.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT 3 instruction. See Chapter 6, “Interrupt 3—Breakpoint Exception (#BP).” Debuggers use break exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

17.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify **transactional regions** (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does **not** resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered.

(A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

17.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt and exception recording:

— Section 17.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)"

— Section 17.6, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture"

— Section 17.8, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem"

— Section 17.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge"

— Section 17.10, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture"

— Section 17.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture"

— Section 17.13, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)"

— Section 17.14, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"

— Section 17.15, "Last Branch, Interrupt, and Exception Recording (P6 Family Processors)"

The following subsections of Section 17.4 describe common features of profiling branches. These features are generally enabled using the IA32_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR_DEBUGCTLA, MSR_DEBUGCTLB, MSR_DEBUGCTL).

17.4.1 IA32_DEBUGCTL MSR

The IA32_DEBUGCTL MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 17.5.1, "LBR Stack" (Intel® Core™2 Duo and Intel® Atom™ Processor Family) and Section 17.8.1, "LBR Stack" (processors based on Intel® Microarchitecture code name Nehalem).
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.

- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."
- **BTINT (branch trace interrupt) flag (bit 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

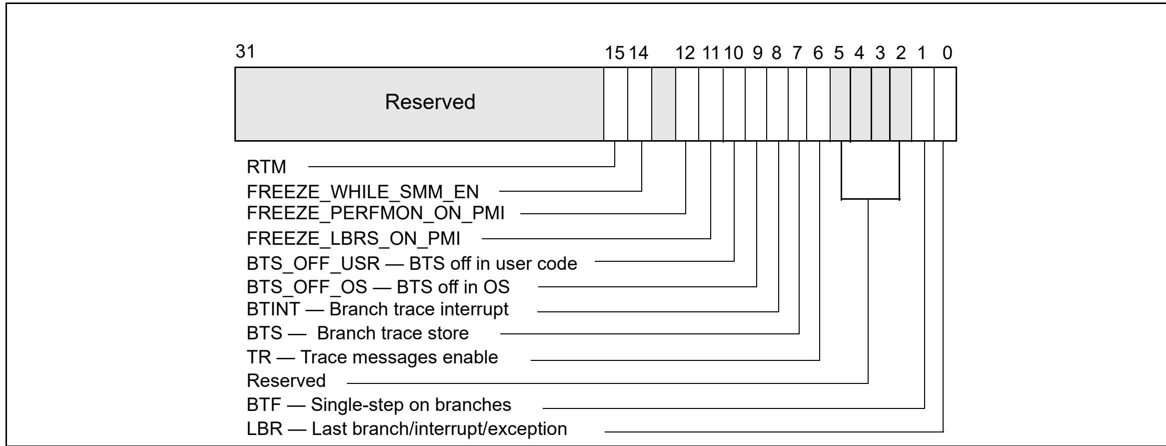


Figure 17-3. IA32_DEBUGCTL MSR for Processors based on Intel Core microarchitecture

- **BTS_OFF_OS (branch trace off in privileged code) flag (bit 9)** — When set, BTS or BTM is skipped if CPL is 0. See Section 17.12.2.
- **BTS_OFF_USR (branch trace off in user code) flag (bit 10)** — When set, BTS or BTM is skipped if CPL is greater than 0. See Section 17.12.2.
- **FREEZE_LBRS_ON_PMI flag (bit 11)** — When set, the LBR stack is frozen on a hardware PMI request (e.g. when a counter overflows and is configured to trigger PMI). See Section 17.4.7 for details.
- **FREEZE_PERFMON_ON_PMI flag (bit 12)** — When set, the performance counters (IA32_PMCx and IA32_FIXED_CTRx) are frozen on a PMI request. See Section 17.4.7 for details.
- **FREEZE_WHILE_SMM_EN (bit 14)** — If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler. Subsequently, the enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. Note that system software must check if the processor supports the IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN control bit. IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.19 for details of detecting the presence of IA32_PERF_CAPABILITIES MSR.
- **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 17.3.3.

17.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

17.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.[1] This mechanism allows a debugger to single-step on control transfers caused by branches. This "branch single stepping" helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

17.4.4 Branch Trace Messages

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

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 17.4.2, "Monitoring Branches, Exceptions, and Interrupts."

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

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

17.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

17.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 17.4.9.2, "Setting Up the DS Save Area." for additional details.

---

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, "Task-State Segment (TSS)."

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

17.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if CPUID.01H:ECX[bit 4] = 1.

The CPL-qualified branch trace mechanism is described in Section 17.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, BTS_OFF_USR (bit 10) and BTS_OFF_OS (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

17.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e. CPUID.0AH:EAX[7:0] > 1). These capabilities provides the following interface in IA32_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - — Legacy Freeze_LBR_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32_DEBUGCTL. Software must then re-enable IA32_DEBUGCTL.LBR to resume recording branches. When using this feature, software should be careful about writes to IA32_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - — Streamlined Freeze_LBR_on_PMI is supported for ArchPerfMonVerID >= 4. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.LBR_Frz =1 to disable recording, but does not change the LBR bit (bit 0) in IA32_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI** (bit 12) — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - — Legacy Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_Perfmon_On_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32_PERF_GLOBAL_CTRL MSR (see Figure 18-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 18.4.1, “Fixed-function Performance Counters”). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32_PERF_GLOBAL_CTRL before leaving a PMI service routine to continue counter operation.
  - — Streamlined Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.CTR_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32_PERF_GLOBAL_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

- A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  - — For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32_PERFEVTSELx register.

  — For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR_PERF_FIXED_CTR_CTRL register (see Figure 18-1) or IA32_FIXED_CTR_CTRL MSR (see Figure 18-2).

- The PEBS buffer is almost full and reaches the interrupt threshold.
- The BTS buffer is almost full and reaches the interrupt threshold.

Table 17-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeza_Perfmon_On_PMI interface.

Table 17-3. Legacy and Streamlined Operation with Freeze_Perfmon_On_PMI = 1, Counter Overflowed

Legacy Freeze_Perfmon_On_PMI	Streamlined Freeze_Perfmon_On_PMI	Comment
Processor freezes the counters on overflow	Processor freezes the counters on overflow	Unchanged
Processor clears IA32_PERF_GLOBAL_CTRL	Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ	
Handler reads IA32_PERF_GLOBAL_STATUS (0x38E) to examine which counter(s) overflowed	**mask** = RDMSR(0x38E)	Similar
Handler services the PMI	Handler services the PMI	Unchanged
Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL (0x390)	Handler writes **mask** into IA32_PERF_GLOBAL_OVF_RESET (0x390)	
Processor clears IA32_PERF_GLOBAL_STATUS	Processor clears IA32_PERF_GLOBAL_STATUS	Unchanged
Handler re-enables IA32_PERF_GLOBAL_CTRL	None	Reduced software overhead

17.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 17-4 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see CPUID instruction in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 17-4. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Component of an LBR Entry	Range of TOS Pointer
06_5CH, 06_5FH	32	FROM_IP, TO_IP	0 to 31
06_4EH, 06_5EH, 06_8EH, 06_9EH	32	FROM_IP, TO_IP, LBR_INFO[1]	0 to 31
06_3DH, 06_47H, 06_4FH, 06_56H	16	FROM_IP, TO_IP	0 to 15
06_3CH, 06_45H, 06_46H, 06_3FH	16	FROM_IP, TO_IP	0 to 15
06_2AH, 06_2DH, 06_3AH, 06_3EH	16	FROM_IP, TO_IP	0 to 15
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	16	FROM_IP, TO_IP	0 to 15
06_17H, 06_1DH	4	FROM_IP, TO_IP	0 to 3
06_0FH	4	FROM_IP, TO_IP	0 to 3
06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH	8	FROM_IP, TO_IP	0 to 7
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	8	FROM_IP, TO_IP	0 to 7

NOTES:

1. See Section 17.11.

The last branch recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults). The branch recording mechanisms generally employs a set of MSRs, referred to as last branch record (LBR) stack. The size and exact locations of the LBR stack are generally model-specific (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for model-specific MSR addresses).

- **Last Branch Record (LBR) Stack —** The LBR consists of N pairs of MSRs (N is listed in the LBR stack size column of Table 17-4) that store source and destination address of recent branches (see Figure 17-3):
  - — MSR_LASTBRANCH_0_FROM_IP (address is model specific) through the next consecutive (N-1) MSR address store source addresses.
  - — MSR_LASTBRANCH_0_TO_IP (address is model specific) through the next consecutive (N-1) MSR address store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant M bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address is model specific) contains an M-bit pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. The valid range of the M-bit POS pointer is given in Table 17-4.

17.4.8.1 LBR Stack and Intel® 64 Processors

LBR MSRs are 64-bits. In 64-bit mode, last branch records store the full address. Outside of 64-bit mode, the upper 32-bits of branch addresses will be stored as 0.

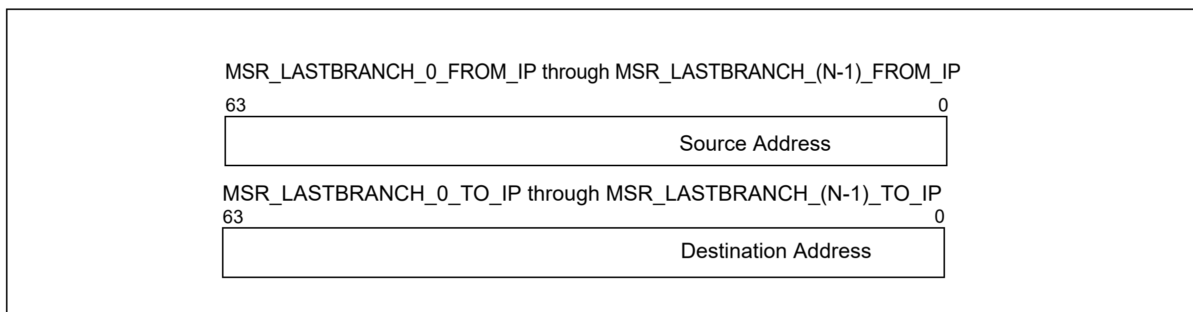


Figure 17-4. 64-bit Address Layout of LBR MSR

Software should query an architectural MSR IA32_PERF_CAPABILITIES[5:0] about the format of the address that is stored in the LBR stack. Four formats are defined by the following encoding:

- — **000000B (32-bit record format) —** Stores 32-bit offset in current CS of respective source/destination,
- — **000001B (64-bit LIP record format) —** Stores 64-bit linear address of respective source/destination,
- — **000010B (64-bit EIP record format) —** Stores 64-bit offset (effective address) of respective source/destination.
- — **000011B (64-bit EIP record format) and Flags** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction info is reported in the upper bit of 'FROM' registers in the LBR stack. See LBR stack details below for flag support and definition.
- — **000100B (64-bit EIP record format), Flags and TSX** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction and TSX info are reported in the upper bits of 'FROM' registers in the LBR stack.
- — **000101B (64-bit EIP record format), Flags, TSX, LBR_INFO** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.
- — **000110B (64-bit EIP record format), Flags, Cycles** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction info is reported in the upper bits of

'FROM' registers in the LBR stack. Elapsed cycles since the last LBR update are reported in the upper 16 bits of the 'TO' registers in the LBR stack (see Section 17.6).

Processor's support for the architectural MSR IA32_PERF_CAPABILITIES is provided by CPUID.01H:ECX[PERF_CAPAB_MSR] (bit 15).

17.4.8.2 LBR Stack and IA-32 Processors

The LBR MSRs in IA-32 processors introduced prior to Intel 64 architecture store the 32-bit "To Linear Address" and "From Linear Address" using the high and low half of each 64-bit MSR.

17.4.8.3 Last Exception Records and Intel 64 Architecture

Intel 64 and IA-32 processors also provide MSRs that store the branch record for the last branch taken prior to an exception or an interrupt. The location of the last exception record (LER) MSRs are model specific. The MSRs that store last exception records are 64-bits. If IA-32e mode is disabled, only the lower 32-bits of the address is recorded. If IA-32e mode is enabled, the processor writes 64-bit values into the MSR. In 64-bit mode, last exception records store 64-bit addresses; in compatibility mode, the upper 32-bits of last exception records are cleared.

17.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] indicates that the processor provides the debug store (DS) mechanism. The DS mechanism allows:

- BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."
- Processor event-based sampling (PEBS) also uses the DS save area provided by debug store mechanism. The capability of PEBS varies across different microarchitectures. See Section 18.4.4, "Processor Event Based Sampling (PEBS)," and the relevant PEBS sub-sections across the core PMU sections in Chapter 18, "Performance Monitoring.".

When CPUID.1:EDX[21] is set:

- The BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags in the IA32_MISC_ENABLE MSR indicate (when clear) the availability of the BTS and PEBS facilities, including the ability to set the BTS and BTINT bits in the appropriate DEBUGCTL MSR.
- The IA32_DS_AREA MSR exists and points to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records** — When the BTS flag in the IA32_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records** — When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a specified value, and event counting begins again. The content layout of a PEBS record varies across different implementations that support PEBS. See Section 18.4.4.2 for details of enumerating PEBS record format.

NOTES

Prior to processors based on the Goldmont microarchitecture, PEBS facility only supports a subset of implementation-specific precise events. See Section 18.7.1 for a PEBS enhancement that can generate records for both precise and non-precise events.

The DS save area and recording mechanism are disabled on INIT, processor Reset or transition to system-management mode (SMM) or IA-32e mode. It is similarly disabled on the generation of a machine-check exception on 45nm and 32nm Intel Atom processors and on processors with Netburst or Intel Core microarchitecture.

The BTS and PEBS facilities may not be available on all processors. The availability of these facilities is indicated by the BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags, respectively, in the IA32_MISC_ENABLE MSR (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*).

The DS save area is divided into three parts (see Figure 17-5): buffer management area, branch trace store (BTS) buffer, and PEBS buffer. The buffer management area is used to define the location and size of the BTS and PEBS buffers. The processor then uses the buffer management area to keep track of the branch and/or PEBS records in their respective buffers and to record the performance counter reset value. The linear address of the first byte of the DS buffer management area is specified with the IA32_DS_AREA MSR.

The fields in the buffer management area are as follows:

- **BTS buffer base** — Linear address of the first byte of the BTS buffer. This address should point to a natural doubleword boundary.
- **BTS index** — Linear address of the first byte of the next BTS record to be written to. Initially, this address should be the same as the address in the BTS buffer base field.
- **BTS absolute maximum** — Linear address of the next byte past the end of the BTS buffer. This address should be a multiple of the BTS record size (12 bytes) plus 1.
- **BTS interrupt threshold** — Linear address of the BTS record on which an interrupt is to be generated. This address must point to an offset from the BTS buffer base that is a multiple of the BTS record size. Also, it must be several records short of the BTS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the BTS absolute maximum record.
- **PEBS buffer base** — Linear address of the first byte of the PEBS buffer. This address should point to a natural doubleword boundary.
- **PEBS index** — Linear address of the first byte of the next PEBS record to be written to. Initially, this address should be the same as the address in the PEBS buffer base field.
- **PEBS absolute maximum** — Linear address of the next byte past the end of the PEBS buffer. This address should be a multiple of the PEBS record size (40 bytes) plus 1.
- **PEBS interrupt threshold** — Linear address of the PEBS record on which an interrupt is to be generated. This address must point to an offset from the PEBS buffer base that is a multiple of the PEBS record size. Also, it must be several records short of the PEBS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the PEBS absolute maximum record.
- **PEBS counter reset value** — A 40-bit value that the counter is to be reset to after state information has collected following counter overflow. This value allows state information to be collected after a preset number of events have been counted.

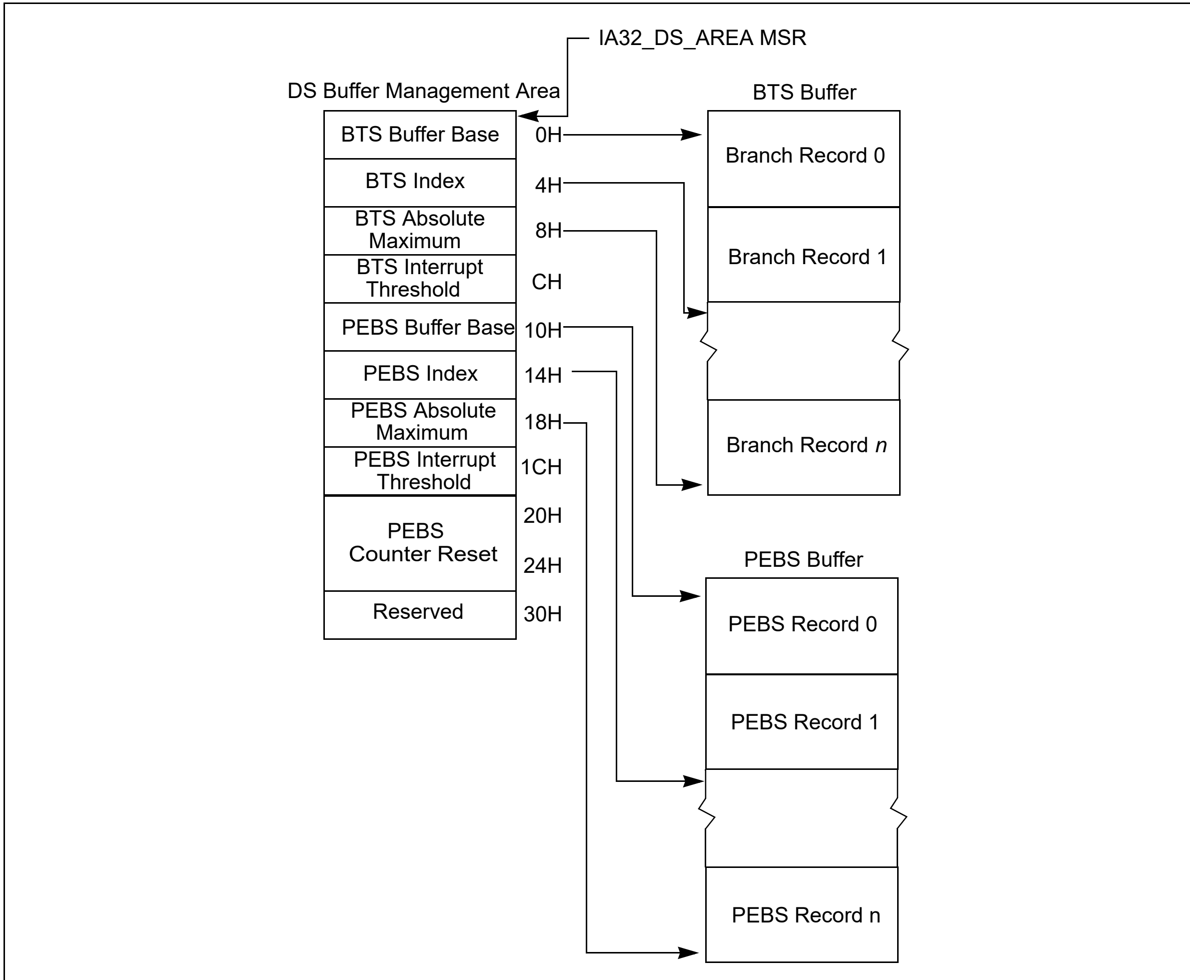


Figure 17-5. DS Save Area

Figures 17-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **Last branch from** — Linear address of the instruction from which the branch, interrupt, or exception was taken.
- **Last branch to** — Linear address of the branch target or the first instruction in the interrupt or exception service routine.
- **Branch predicted** — Bit 4 of field indicates whether the branch that was taken was predicted (set) or not predicted (clear).

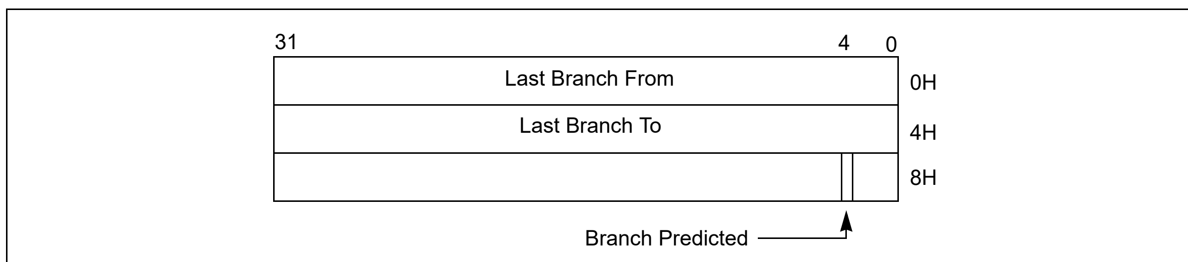


Figure 17-6. 32-bit Branch Trace Record Format

Figure 17-7 shows the structure of the 40-byte PEBS records. Nominally the register values are those at the beginning of the instruction that caused the event. However, there are cases where the registers may be logged in a partially modified state. The linear IP field shows the value in the EIP register translated from an offset into the current code segment to a linear address.

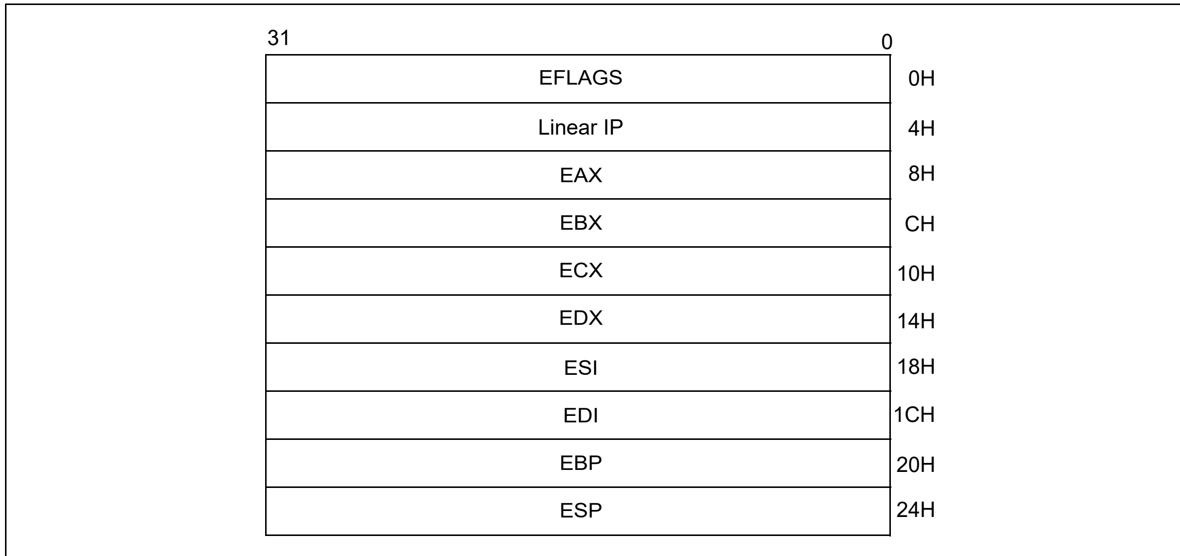


Figure 17-7. PEBS Record Format

17.4.9.1 64 Bit Format of the DS Save Area

When DTES64 = 1 (CPUID.1.ECX[2] = 1), the structure of the DS save area is shown in Figure 17-8.

When DTES64 = 0 (CPUID.1.ECX[2] = 0) and IA-32e mode is active, the structure of the DS save area is shown in Figure 17-8. If IA-32e mode is not active the structure of the DS save area is as shown in Figure 17-6.

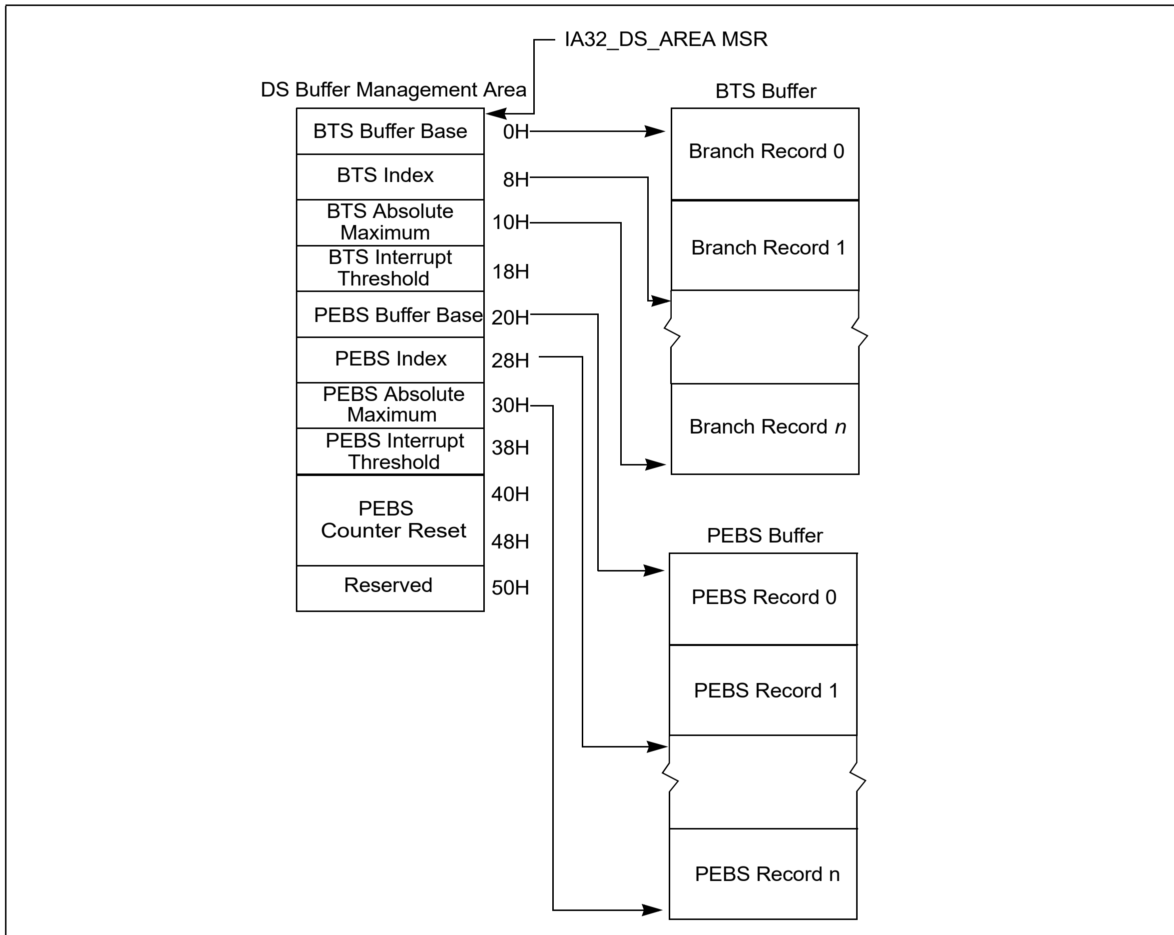


Figure 17-8. IA-32e Mode DS Save Area

The IA32_DS_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 17-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 17-9). The structure of a PEBS record is similar to that shown in Figure 17-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 17-10).

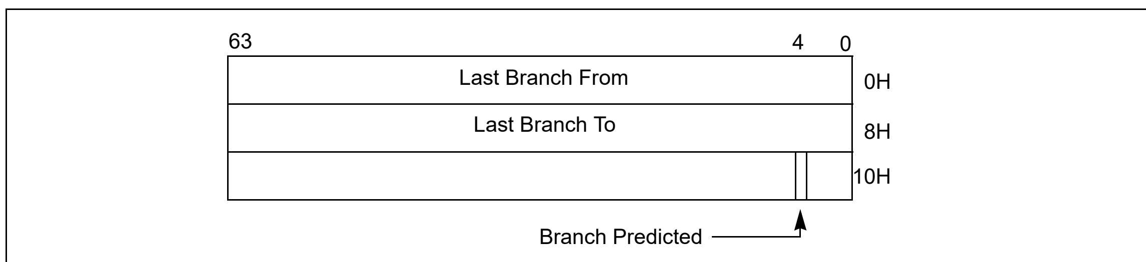


Figure 17-9. 64-bit Branch Trace Record Format

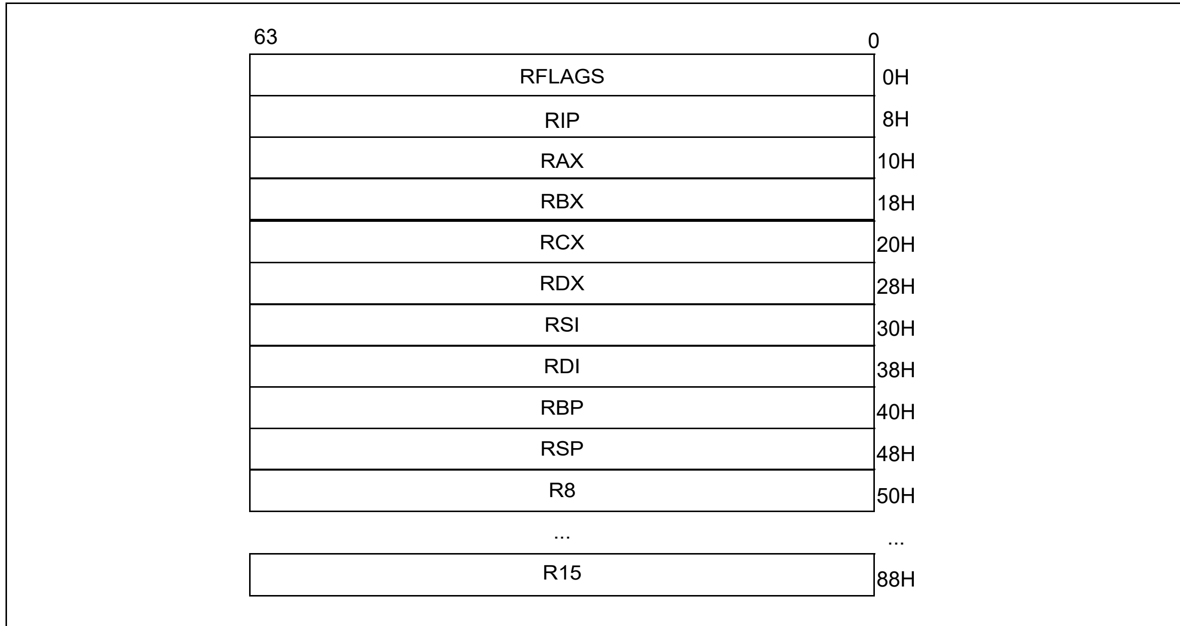


Figure 17-10. 64-bit PEBS Record Format

Fields in the buffer management area of a DS save area are described in Section 17.4.9.

The format of a branch trace record and a PEBS record are the same as the 64-bit record formats shown in Figures 17-9 and Figures 17-10, with the exception that the branch predicted bit is not supported by Intel Core microarchitecture or Intel Atom microarchitecture. The 64-bit record formats for BTS and PEBS apply to DS save area for all operating modes.

The procedures used to program IA32_DEBUGCTL MSR to set up a BTS buffer or a CPL-qualified BTS are described in Section 17.4.9.3 and Section 17.4.9.4.

Required elements for writing a DS interrupt service routine are largely the same on processors that support using DS Save area for BTS or PEBS records. However, on processors based on Intel NetBurst® microarchitecture, re-enabling counting requires writing to CCCRs. But a DS interrupt service routine on processors supporting architectural performance monitoring should:

- Re-enable the enable bits in IA32_PERF_GLOBAL_CTRL MSR if it is servicing an overflow PMI due to PEBS.
- Clear overflow indications by writing to IA32_PERF_GLOBAL_OVF_CTRL when a counting configuration is changed. This includes bit 62 (ClrOvfBuffer) and the overflow indication of counters used in either PEBS or general-purpose counting (specifically: bits 0 or 1; see Figures 18-3).

17.4.9.2 Setting Up the DS Save Area

To save branch records with the BTS buffer, the DS save area must first be set up in memory as described in the following procedure (See Section 18.4.4.1, "Setting up the PEBS Buffer," for instructions for setting up a PEBS buffer, respectively, in the DS save area):

1. Create the DS buffer management information area in memory (see Section 17.4.9, "BTS and DS Save Area," and Section 17.4.9.1, "64 Bit Format of the DS Save Area"). Also see the additional notes in this section.
2. Write the base linear address of the DS buffer management area into the IA32_DS_AREA MSR.
3. Set up the performance counter entry in the xAPIC LVT for fixed delivery and edge sensitive. See Section 10.5.1, "Local Vector Table."
4. Establish an interrupt handler in the IDT for the vector associated with the performance counter entry in the xAPIC LVT.

5. Write an interrupt service routine to handle the interrupt. See Section 17.4.9.5, “Writing the DS Interrupt Service Routine.”

The following restrictions should be applied to the DS save area.

- The three DS save area sections should be allocated from a non-paged pool, and marked accessed and dirty. It is the responsibility of the operating system to keep the pages that contain the buffer present and to mark them accessed and dirty. The implication is that the operating system cannot do “lazy” page-table entry propagation for these pages.
- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.
- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.
- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.
- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.
- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.
- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.
- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.
- The DS save area is expected to used only on systems with an enabled APIC. The LVT Performance Counter entry in the APCI must be initialized to use an interrupt gate instead of the trap gate.

17.4.9.3 Setting Up the BTS Buffer

Three flags in the MSR_DEBUGCTLA MSR (see Table 17-5), IA32_DEBUGCTL (see Figure 17-3), or MSR_DEBUGCTLB (see Figure 17-16) control the generation of branch records and storing of them in the BTS buffer; these are TR, BTS, and BTINT. The TR flag enables the generation of BTMs. The BTS flag determines whether the BTMs are sent out on the system bus (clear) or stored in the BTS buffer (set). BTMs cannot be simultaneously sent to the system bus and logged in the BTS buffer. The BTINT flag enables the generation of an interrupt when the BTS buffer is full. When this flag is clear, the BTS buffer is a circular buffer.

Table 17-5. IA32_DEBUGCTL Flag Encodings

TR	BTS	BTINT	Description
0	X	X	Branch trace messages (BTMs) off
1	0	X	Generate BTMs
1	1	0	Store BTMs in the BTS buffer, used here as a circular buffer
1	1	1	Store BTMs in the BTS buffer, and generate an interrupt when the buffer is nearly full

The following procedure describes how to set up a DS Save area to collect branch records in the BTS buffer:

1. Place values in the BTS buffer base, BTS index, BTS absolute maximum, and BTS interrupt threshold fields of the DS buffer management area to set up the BTS buffer in memory.
2. Set the TR and BTS flags in the IA32_DEBUGCTL for Intel Core Solo and Intel Core Duo processors or later processors (or MSR_DEBUGCTLA MSR for processors based on Intel NetBurst Microarchitecture; or MSR_DEBUGCTLB for Pentium M processors).
3. Clear the BTINT flag in the corresponding IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR; or MSR_DEBUGCTLB) if a circular BTS buffer is desired.

NOTES

If the buffer size is set to less than the minimum allowable value (i.e. BTS absolute maximum < 1 + size of BTS record), the results of BTS is undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

17.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS_OFF_OS, BTS_OFF_USR, and BTINT. The encoding of these five bits are shown in Table 17-6.

Table 17-6. CPL-Qualified Branch Trace Store Encodings

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

17.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.

  BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.
- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR_PEBS_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.
- The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.

- After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.
- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the counters to count via IA32_PERF_GLOBAL_CTRL/IA32_PERF_GLOBAL_OVF_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR's ENABLE bit on processor based on Intel NetBurst microarchitecture).
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

17.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL® ATOM™ PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

- **Debug Trace and Branch Recording Control** — The IA32_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-3 for the MSR layout.
- **Last branch record (LBR) stack** — There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.5.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts**
  - See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  - 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE_LBRS_ON_PMI flag is set.
- **Branch trace messages** — See Section 17.4.4.
- **Last exception records** — See Section 17.12.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **FREEZE_WHILE_SMM_EN (bit 14)** — FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 17.4.1.

17.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_3_FROM_IP (address 43H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_3_TO_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_7_FROM_IP (address 47H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_7_TO_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

17.5.2 LBR Stack in Intel Atom Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 17-11.

17.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Processors based on the Goldmont microarchitecture extend the capabilities described in Section 17.5.2 with the following enhancements:

- Supports new LBR format encoding 00110b in IA32_PERF_CAPABILITIES[5:0].
- Size of LBR stack increased to 32. Each entry includes MSR_LASTBRANCH_x_FROM_IP (address 0x680..0x69f) and MSR_LASTBRANCH_x_TO_IP (address 0x6c0..0x6df).
- LBR call stack filtering supported. The layout of MSR_LBR_SELECT is described in Table 17-13.
- Elapsed cycle information is added to MSR_LASTBRANCH_x_TO_IP. Format is shown in Table 17-7.
- Misprediction info is reported in the upper bits of MSR_LASTBRANCH_x_FROM_IP. MISPRED bit format is shown in Table 17-8.
- Streamlined Freeze_LBRs_On_PMI operation; see Section 17.11.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 17.11.3.

Table 17-7. MSR_LASTBRANCH_x_TO_IP for the Goldmont Microarchitecture

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch to" address. See Section 17.4.8.1 for address format.
Cycle Count (Saturating)	63:48	R/W	Elapsed core clocks since last update to the LBR stack.

17.7 LAST BRANCH, INTERRUPT AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 680H) through MSR_LASTBRANCH_7_FROM_IP (address 687H) store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address 6C0H) through MSR_LASTBRANCH_7_TO_IP (address 6C7H) store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 17-11.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 17.4.8.1.The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

17.8 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

The processors based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and adds additional capabilities:

- **Debug Trace and Branch Recording Control** — The IA32_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-11 for the MSR layout.
- **Last branch record (LBR) stack** — There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.8.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts** — See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
- **Branch trace messages** — The IA32_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 17.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records** — See Section 17.12.3.
- **Branch trace store and CPL-qualified BTS** — See Section 17.4.6 and Section 17.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11)** — see Section 17.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12)** — see Section 17.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **UNCORE_PMI_EN (bit 13)** — When set. this logical processor is enabled to receive an counter overflow interrupt form the uncore.
- **FREEZE_WHILE_SMM_EN (bit 14)** — FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 17.4.1.

Processors based on Intel microarchitecture code name Nehalem provide additional capabilities:

- **Independent control of uncore PMI** — The IA32_DEBUGCTL MSR provides a bit field (see Figure 17-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.
- **LBR filtering** — Processors based on Intel microarchitecture code name Nehalem support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT.

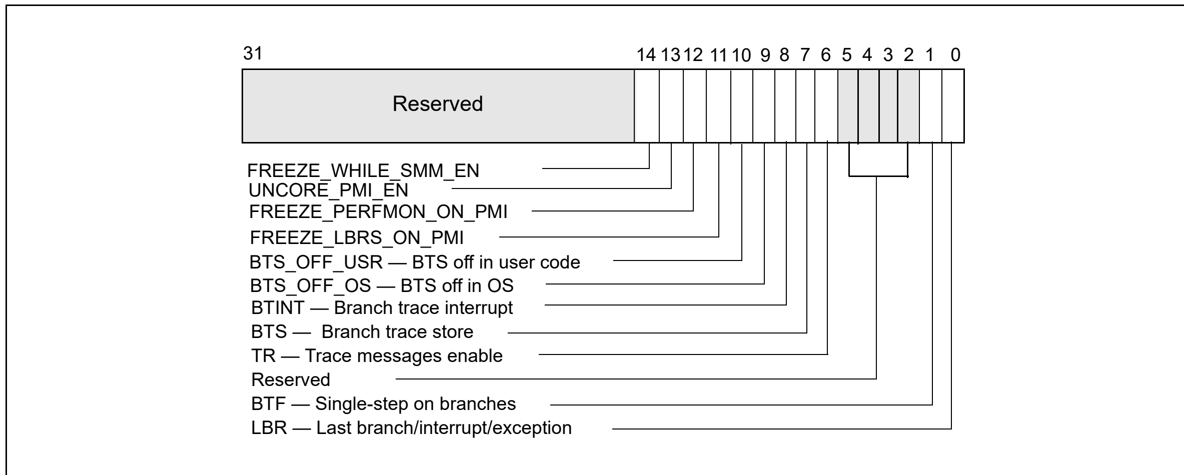


Figure 17-11. IA32_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem

17.8.1 LBR Stack

Processors based on Intel microarchitecture code name Nehalem provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 17-8 and Table 17-9.

Table 17-8. MSR_LASTBRANCH_x_FROM_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch from” address. See Section 17.4.8.1 for address format.
SIGN_EXt	62:48	R/W	Signed extension of bit 47 of this register.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

Table 17-9. MSR_LASTBRANCH_x_TO_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the “branch to” address. See Section 17.4.8.1 for address format
SIGN_EXt	63:48	R/W	Signed extension of bit 47 of this register.

Processors based on Intel microarchitecture code name Nehalem have an LBR MSR Stack as shown in Table 17-10.

Table 17-10. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_1AH	16	0 to 15

17.8.2 Filtering of Last Branch Records

MSR_LBR_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e. all branches will be captured. MSR_LBR_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR_LBR_SELECT is shown in Table 17-11.

Table 17-11. MSR_LBR_SELECT for Intel microarchitecture code name Nehalem

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

17.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.8, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem", apply to processors based on Intel microarchitecture code name Sandy Bridge. For processors based on Intel microarchitecture code name Ivy Bridge, the same holds true.

One difference of note is that MSR_LBR_SELECT is shared between two logical processors in the same core. In Intel microarchitecture code name Sandy Bridge, each logical processor has its own MSR_LBR_SELECT. The filtering semantics for "Near_ind_jmp" and "Near_rel_jmp" has been enhanced, see Table 17-12.

Table 17-12. MSR_LBR_SELECT for Intel® microarchitecture code name Sandy Bridge

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns

Table 17-12. MSR_LBR_SELECT for Intel® microarchitecture code name Sandy Bridge (Contd.)

Bit Field	Bit Offset	Access	Description
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

17.10 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge", apply to next generation processors based on Intel microarchitecture code name Haswell.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR_LBR_SELECT.EN_CALLSTACK[bit 9]; see Table 17-13. If MSR_LBR_SELECT.EN_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 17.9.

Table 17-13. MSR_LBR_SELECT for Intel® microarchitecture code name Haswell

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
EN_CALLSTACK[1]	9		Enable LBR stack to use LIFO filtering to capture Call stack profile
Reserved	63:10		Must be zero

NOTES:

1. Must set valid combination of bits 0-8 in conjunction with bit 9 (as described below), otherwise the contents of the LBR MSRs are undefined.

The call stack profiling capability is an enhancement of the LBR facility. The LBR stack is a ring buffer typically used to profile control flow transitions resulting from branches. However, the finite depth of the LBR stack often become less effective when profiling certain high-level languages (e.g. C++), where a transition of the execution flow is accompanied by a large number of leaf function calls, each of which returns an individual parameter to form the list of parameters for the main execution function call. A long list of such parameters returned by the leaf functions would serve to flush the data captured in the LBR stack, often losing the main execution context.

When the call stack feature is enabled, the LBR stack will capture unfiltered call data normally, but as return instructions are executed the last captured branch record is flushed from the on-chip registers in a last-in first-out (LIFO) manner. Thus, branch information relative to leaf functions will not be captured, while preserving the call stack information of the main line execution path.

The configuration of the call stack facility is summarized below:

- Set IA32_DEBUGCTL.LBR (bit 0) to enable the LBR stack to capture branch records. The source and target addresses of the call branches will be captured in the 16 pairs of From/To LBR MSRs that form the LBR stack.

- Program the Top of Stack (TOS) MSR that points to the last valid from/to pair. This register is incremented by 1, modulo 16, before recording the next pair of addresses.
- Program the branch filtering bits of MSR_LBR_SELECT (bits 0:8) as desired.
- Program the MSR_LBR_SELECT to enable LIFO filtering of return instructions with:
  — The following bits in MSR_LBR_SELECT must be set to '1': JCC, NEAR_IND_JMP, NEAR_REL_JMP, FAR_BRANCH, EN_CALLSTACK;
  — The following bits in MSR_LBR_SELECT must be cleared: NEAR_REL_CALL, NEAR-IND_CALL, NEAR_RET;
  — At most one of CPL_EQ_0, CPL_NEQ_0 is set.

Note that when call stack profiling is enabled, "zero length calls" are excluded from writing into the LBRs. (A "zero length call" uses the attribute of the call instruction to push the immediate instruction pointer on to the stack and then pops off that address into a register. This is accomplished without any matching return on the call.)

17.10.1 LBR Stack Enhancement

Processors based on Intel microarchitecture code name Haswell provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is enumerated by IA32_PERF_CAPABILITIES[5:0] = 04H, and is shown in Table 17-14 and Table 17-9.

Table 17-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch from" address. See Section 17.4.8.1 for address format.
SIGN_EXT	60:48	R/W	Signed extension of bit 47 of this register.
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region, or EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

17.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE

Processors based on the Skylake microarchitecture provide a number of enhancement with storing last branch records:

- enumeration of new LBR format: encoding 00101b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of a triplets of MSRs:
  — MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 17-9.
  — MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 17-9.
  — MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data.
- Size of LBR stack increased to 32.

Processors based on the Skylake microarchitecture supports the same LBR filtering capabilities as described in Table 17-13.

Table 17-15. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_4EH, 06_5EH	32	0 to 31

17.11.1 MSR_LBR_INFO_x MSR

The layout of each MSR_LBR_INFO_x MSR is shown in Table 17-16.

Table 17-16. MSR_LBR_INFO_x

Bit Field	Bit Offset	Access	Description
Cycle Count (saturating)	15:0	R/W	Elapsed core clocks since last update to the LBR stack.
Reserved	60:16	R/W	Reserved
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region OR EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

17.11.2 Streamlined Freeze_LBRs_On_PMI Operation

The FREEZE_LBRS_ON_PMI feature causes the LBRs to be frozen on a hardware request for a PMI. This prevents the LBRs from being overwritten by new branches, allowing the PMI handler to examine the control flow that preceded the PMI generation. Architectural performance monitoring version 4 and above supports a streamlined FREEZE_LBRs_ON_PMI operation for PMI service routine that replaces the legacy FREEZE_LBRs_ON_PMI operation (see Section 17.4.7).

While the legacy FREEZE_LBRS_ON_PMI clear the LBR bit in the IA32_DEBUGCTL MSR on a PMI request, the streamlined FREEZE_LBRS_ON_PMI will set the LBR_FRZ bit in IA32_PERF_GLOBAL_STATUS. Branches will not cause the LBRs to be updated when LBR_FRZ is set. Software can clear LBR_FRZ at the same time as it clears overflow bits by setting the LBR_FRZ bit as well as the needed overflow bit when writing to IA32_PERF_GLOBAL_STATUS_RESET MSR.

This streamlined behavior avoids race conditions between software and processor writes to IA32_DEBUGCTL that are possible with FREEZE_LBRS_ON_PMI clearing of the LBR enable.

17.11.3 LBR Behavior and Deep C-State

When MWAIT is used to request a C-state that is numerically higher than C1, then LBR state may be initialized to zero depending on optimized "waiting" state that is selected by the processor The affected LBR states include the FROM, TO, INFO, LAST_BRANCH, LER and LBR_TOS registers. The LBR enable bit and LBR_FROZEN bit are not affected. The LBR-time of the first LBR record inserted after an exit from such a C-state request will be zero.

17.12 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

Pentium 4 and Intel Xeon processors based on Intel NetBurst microarchitecture provide the following methods for recording taken branches, interrupts and exceptions:

- Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address.
- Send the branch records out on the system bus as branch trace messages (BTMs).
- Log BTMs in a memory-resident branch trace store (BTS) buffer.

To support these functions, the processor provides the following MSRs and related facilities:

- **MSR_DEBUGCTLA MSR** — Enables last branch, interrupt, and exception recording; single-stepping on taken branches; branch trace messages (BTMs); and branch trace store (BTS). This register is named DebugCtlMSR in the P6 family processors.
- **Debug store (DS) feature flag (CPUID.1:EDX.DS[bit 21])** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer.
- **CPL-qualified debug store (DS) feature flag (CPUID.1:ECX.DS-CPL[bit 4])** — Indicates that the processor provides a CPL-qualified debug store (DS) mechanism, which allows software to selectively skip sending and storing BTMs, according to specified current privilege level settings, into a memory-resident BTS buffer.
- **IA32_MISC_ENABLE MSR** — Indicates that the processor provides the BTS facilities.
- **Last branch record (LBR) stack** — The LBR stack is a circular stack that consists of four MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_15_FROM_IP and MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_15_TO_IP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- **Last branch record top-of-stack (TOS) pointer** — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H]. See also: Table 17-17, Figure 17-12, and Section 17.12.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."
- **Last exception record** — See Section 17.12.3, "Last Exception Records."

17.12.1 MSR_DEBUGCTLA MSR

The MSR_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-12 shows the flags in the MSR_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example, when an instruction or data breakpoint or a single-step trap occurs). See Section 17.12.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."
- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches."

- **TR (trace message enable) flag (bit 2)** — When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, “Branch Trace Messages.”

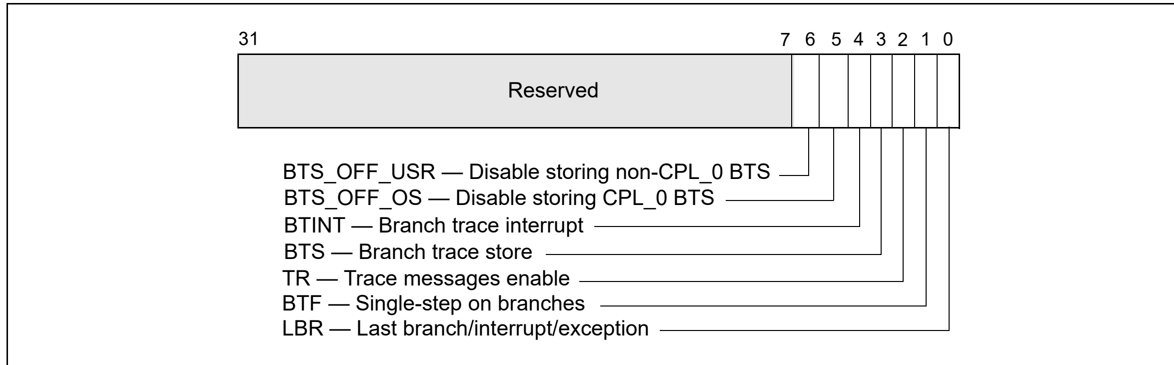


Figure 17-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **BTS (branch trace store) flag (bit 3)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, “BTS and DS Save Area.”
- **BTINT (branch trace interrupt) flag (bits 4)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, “Branch Trace Store (BTS).”
- **BTS_OFF_OS (disable ring 0 branch trace store) flag (bit 5)** — When set, enables the BTS facilities to skip sending/logging CPL_0 BTMs to the memory-resident BTS buffer. See Section 17.12.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **BTS_OFF_USR (disable ring 0 branch trace store) flag (bit 6)** — When set, enables the BTS facilities to skip sending/logging non-CPL_0 BTMs to the memory-resident BTS buffer. See Section 17.12.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

NOTE

The initial implementation of BTS_OFF_USR and BTS_OFF_OS in MSR_DEBUGCTLA is shown in Figure 17-12. The BTS_OFF_USR and BTS_OFF_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

17.12.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR_LASTBRANCH_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches it maximum value, it wraps around to 0. See Table 17-17 and Figure 17-12.

Table 17-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR_LASTBRANCH_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 17-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the "from" and "to" instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the "from" address is the address of the branch instruction and the "to" address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the "from" address the return instruction pointer (RIP) saved for the interrupt and the "to" address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the "from" address is the linear address of the instruction that caused the exception to be generated and the "to" address is the address of the first instruction in the exception handler routine.

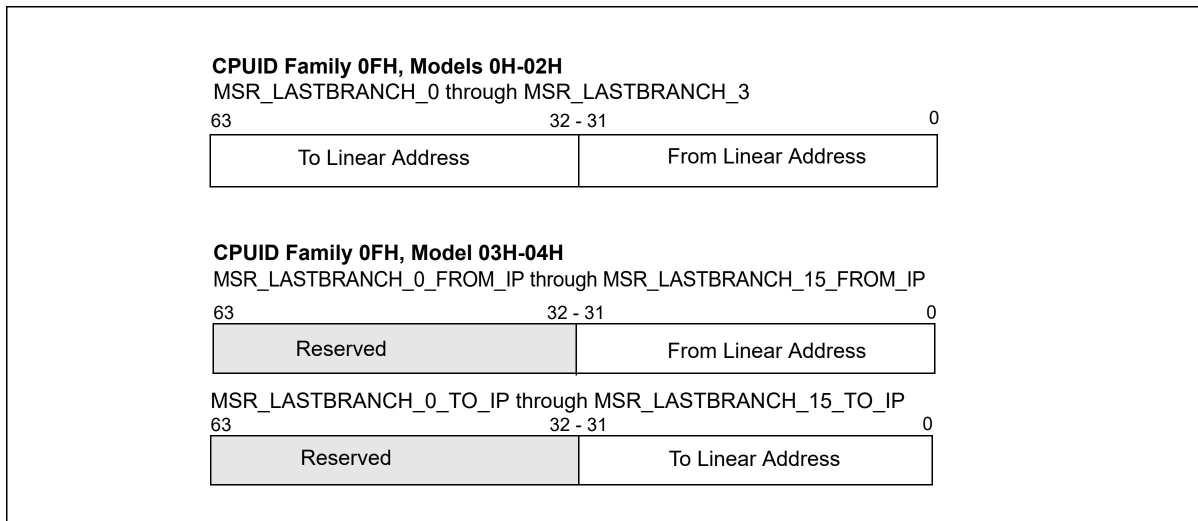


Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

17.12.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors provide two MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that

duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors. The MSR_LER_TO_LIP and MSR_LER_FROM_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

17.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

- **IA32_DEBUGCTL MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32_DEBUGCTL MSR is located at register address 01D9H.

  See Figure 17-14 for the layout and the entries below for a description of the flags:

  — **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" below.

  — **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.

  — **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.

  — **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."

  — **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

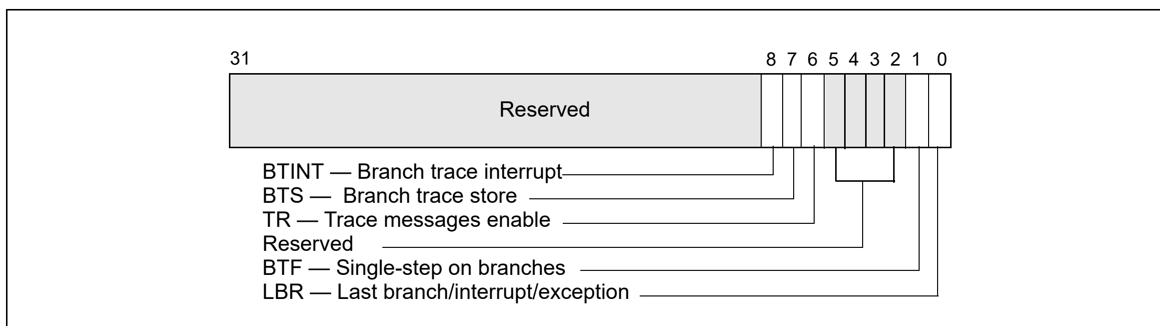


Figure 17-14. IA32_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address (MSR addresses start at 40H). See Figure 17-15.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

For details, see Section 17.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture," and Section 2.19, "MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

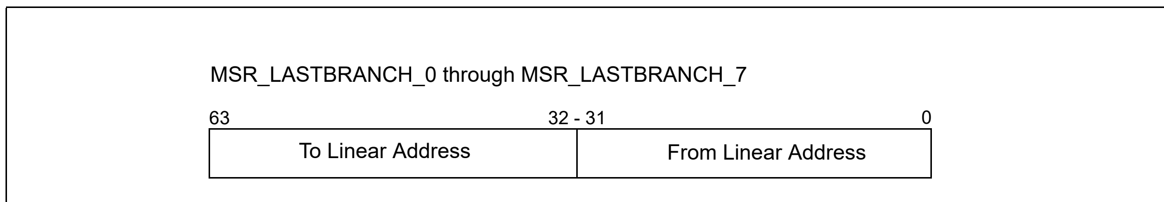


Figure 17-15. LBR Branch Record Layout for the Intel Core Solo and Intel Core Duo Processor

17.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- **MSR_DEBUGCTLB MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 17-16 and the entries below for a description of the flags.
  - **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" bullet below.
  - **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
  - **PB*i* (performance monitoring/breakpoint pins) flags (bits 5-2)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.

— **BTS (branch trace store) flag (bit 7)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."

— **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

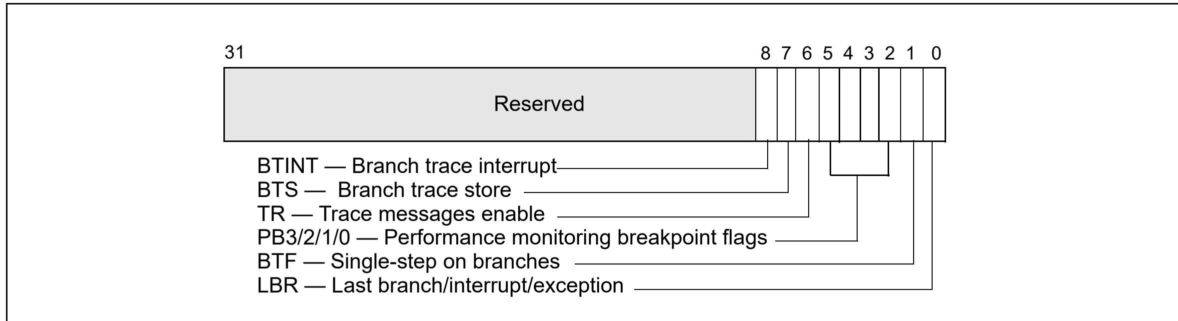


Figure 17-16. MSR_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 17-17.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

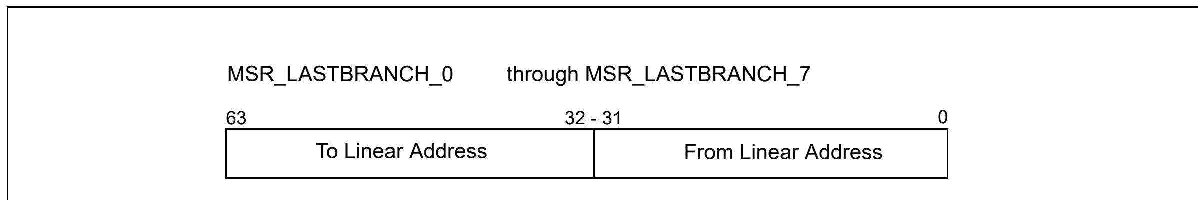


Figure 17-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 17.12.3, "Last Exception Records," and Section 2.20, "MSRs In the Pentium M Processor" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

17.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS)

The P6 family processors provide five MSRs for recording the last branch, interrupt, or exception taken by the processor: DEBUGCTLMSR, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers can be used to collect last branch records, to set breakpoints on branches, interrupts, and exceptions, and to single-step from one branch to the next.

See Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

17.15.1 DEBUGCTLMSR Register

The version of the DEBUGCTLMSR register found in the P6 family processors enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-18 shows the flags in the DEBUGCTLMSR register for the P6 family processors. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records the source and target addresses (in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs) for the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. The processor clears this flag whenever a debug exception, such as an instruction or data breakpoint or single-step trap occurs.

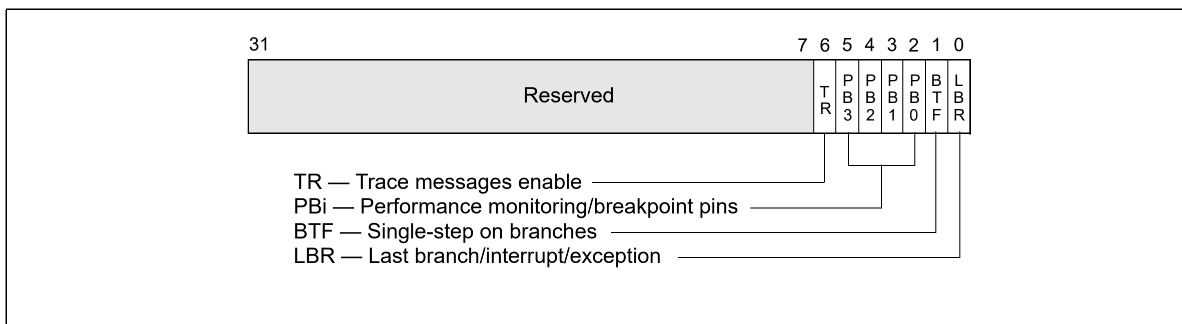


Figure 17-18. DEBUGCTLMSR Register (P6 Family Processors)

- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag. See Section 17.4.3, "Single-Stepping on Branches."
- **PB*i* (performance monitoring/breakpoint pins) flags (bits 2 through 5)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
- **TR (trace message enable) flag (bit 6)** — When set, trace messages are enabled as described in Section 17.4.4, "Branch Trace Messages." Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

17.15.2 Last Branch and Last Exception MSRs

The LastBranchToIP and LastBranchFromIP MSRs are 32-bit registers for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. When a branch occurs, the processor loads the address of the branch instruction into the LastBranchFromIP MSR and loads the target address for the branch into the LastBranchToIP MSR.

When an interrupt or exception occurs (other than a debug exception), the address of the instruction that was interrupted by the exception or interrupt is loaded into the LastBranchFromIP MSR and the address of the exception or interrupt handler that is called is loaded into the LastBranchToIP MSR.

The LastExceptionToIP and LastExceptionFromIP MSRs (also 32-bit registers) record the instruction pointers for the last branch that the processor took prior to an exception or interrupt being generated. When an exception or interrupt occurs, the contents of the LastBranchToIP and LastBranchFromIP MSRs are copied into these registers before the to and from addresses of the exception or interrupt are recorded in the LastBranchToIP and LastBranchFromIP MSRs.

These registers can be read using the RDMSR instruction.

Note that the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into the current code segment, as opposed to linear addresses, which are saved in last branch records for the Pentium 4 and Intel Xeon processors.

17.15.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMSR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the LastBranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

17.16 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter's architecture includes the following components:

- **TSC flag** — A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function CPUID.1:EDX.TSC[bit 4] = 1.
- **IA32_TIME_STAMP_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) — The MSR used as the counter.
- **RDTSC instruction** — An instruction used to read the time-stamp counter.
- **TSD flag** — A control register flag is used to enable or disable the time-stamp counter (enabled if CR4.TSD[bit 2] = 1).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

  The internal processor clock cycle is determined by the current core-clock to bus-clock ratio. Intel® SpeedStep® technology transitions may also impact the processor clock.

- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the processor base frequency, see Section 18.18.2 for more detail. On certain processors, the TSC frequency may not be the same as the frequency in the brand string.

  The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

NOTE

To determine average processor clock frequency, Intel recommends the use of performance monitoring logic to count processor core clocks over the period of time for which the average is required. See Section 18.17, “Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture,” and Chapter 19, “Performance-Monitoring Events,” for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H]], model [0EH, 0FH]; for family [06H]], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

17.16.1 Invariant TSC

The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor’s support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C-. and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

17.16.2 IA32_TSC_AUX Register and RDTSCP Support

Processors based on Intel microarchitecture code name Nehalem provide an auxiliary TSC register, IA32_TSC_AUX that is designed to be used in conjunction with IA32_TSC. IA32_TSC_AUX provides a 32-bit field that is initialized by privileged software with a signature value (for example, a logical processor ID).

The primary usage of IA32_TSC_AUX in conjunction with IA32_TSC is to allow software to read the 64-bit time stamp in IA32_TSC and signature value in IA32_TSC_AUX with the instruction RDTSCP in an atomic operation. RDTSCP returns the 64-bit time stamp in EDX:EAX and the 32-bit TSC_AUX signature value in ECX. The atomicity of RDTSCP ensures that no context switch can occur between the reads of the TSC and TSC_AUX values.

Support for RDTSCP is indicated by CPUID.80000001H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).

User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.

17.16.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32_TIME_STAMP_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way than ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32_TSC_ADJUST MSR (address 3BH). Like the IA32_TIME_STAMP_COUNTER MSR, the IA32_TSC_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32_TSC_ADJUST MSR as follows:

- On RESET, the value of the IA32_TSC_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32_TIME_STAMP_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32_TSC_ADJUST MSR.
- If an execution of WRMSR to the IA32_TSC_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32_TSC_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32_TIME_STAMP_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32_TSC_ADJUST MSR on each logical processor.

Processor support for the IA32_TSC_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC_ADJUST (bit 1).

17.16.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

TSC_Value = (ART_Value * CPUID.15H:EBX[31:0])/ CPUID.15H:EAX[31:0] + K

Where 'K' is an offset that can be adjusted by a privileged agent[2].

When ART hardware is reset, both invariant TSC and K are also reset.

2. IA32_TSC_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time

17.17 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel® Xeon® processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

17.17.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of Resource Monitoring IDs (RMIDs). Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32_PQR_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32_QM_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. Before reading back monitored data software must configure an event selection MSR (IA32_QM_EVTSEL) to specify which metric is to be reported, and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

17.17.2 Enabling Monitoring: Usage Flow

Figure 17-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.

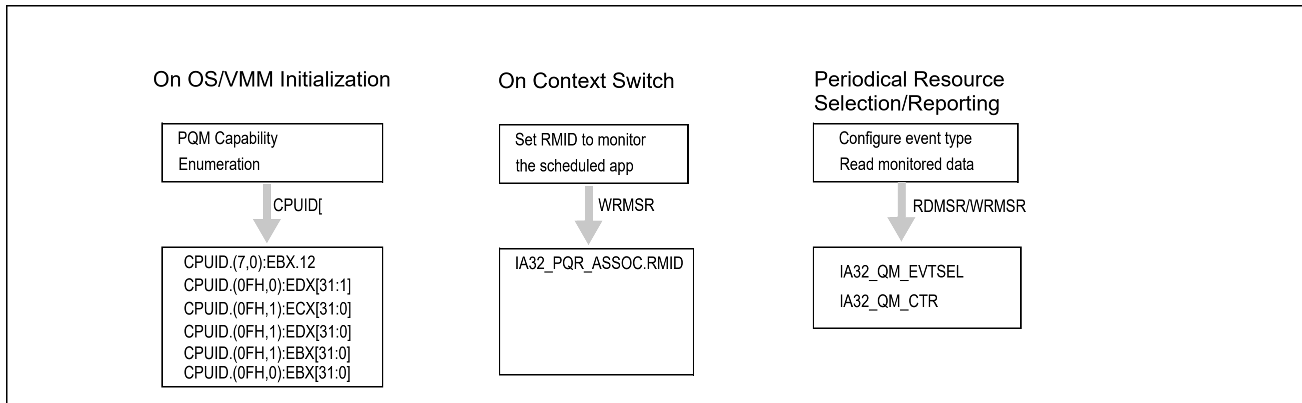


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

17.17.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] reports 1, the processor provides the following programming interfaces for shared resource monitoring, including Cache Monitoring Technology:

- CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides information on available resource types (see Section 17.17.4), and monitoring capabilities for each resource type (see Section 17.17.5). Note CMT and MBM capabilities are enumerated as separate event vectors using shared enumeration infrastructure under a given resource type.
- IA32_PQR_ASSOC.RMID: The per-logical-processor MSR, IA32_PQR_ASSOC, that OS/VMM can use to assign an RMID to each logical processor, see Section 17.17.6.
- IA32_QM_EVTSEL: This MSR specifies an Event ID (EvtID) and an RMID which the platform uses to look up and provide monitoring data in the monitoring counter, IA32_QM_CTR, see Section 17.17.7.
- IA32_QM_CTR: This MSR reports monitored resource data when available along with bits to allow software to check for error conditions and verify data validity.

Software must follow the following sequence of enumeration to discover Cache Monitoring Technology capabilities:

1. Execute CPUID with EAX=0 to discover the “cpuid_maxLeaf” supported in the processor;
2. If cpuid_maxLeaf >= 7, then execute CPUID with EAX=7, ECX= 0 to verify CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CPUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CPUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CPUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

5. If CPUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CPUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CPUID.(EAX=0FH, ECX=0):EDX.

17.17.4 Monitoring Resource Type and Capability Enumeration

CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides one sub-leaf (sub-function 0) that reports shared enumeration infrastructure, and one or more sub-functions that report feature-specific enumeration data:

- Monitoring leaf sub-function 0 enumerates available resources that support monitoring, i.e. executing CPUID with EAX=0FH and ECX=0H. In the initial implementation, L3 cache is the only resource type available. Each supported resource type is represented by a bit in CPUID.(EAX=0FH, ECX=0):EDX[31:1]. The bit position corresponds to the sub-leaf index (ResID) that software must use to query details of the monitoring capability of that resource type (see Figure 17-21 and Figure 17-22). Reserved bits of CPUID.(EAX=0FH, ECX=0):EDX[31:2] correspond to unsupported sub-leaves of the CPUID.0FH leaf. Additionally, CPUID.(EAX=0FH, ECX=0H):EBX reports the highest RMID value of any resource type that supports monitoring in the processor.

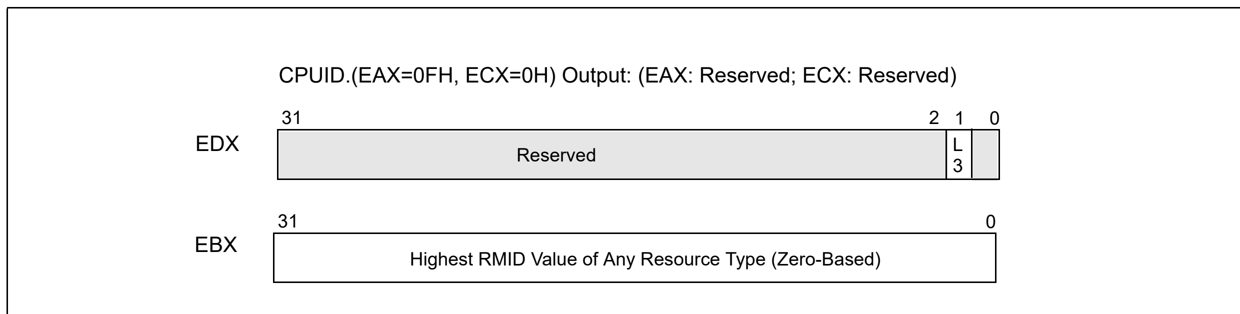


Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

17.17.5 Feature-Specific Enumeration

Each additional sub-leaf of CPUID.(EAX=0FH, ECX=ResID) enumerates the specific details for software to program Monitoring MSRs using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.

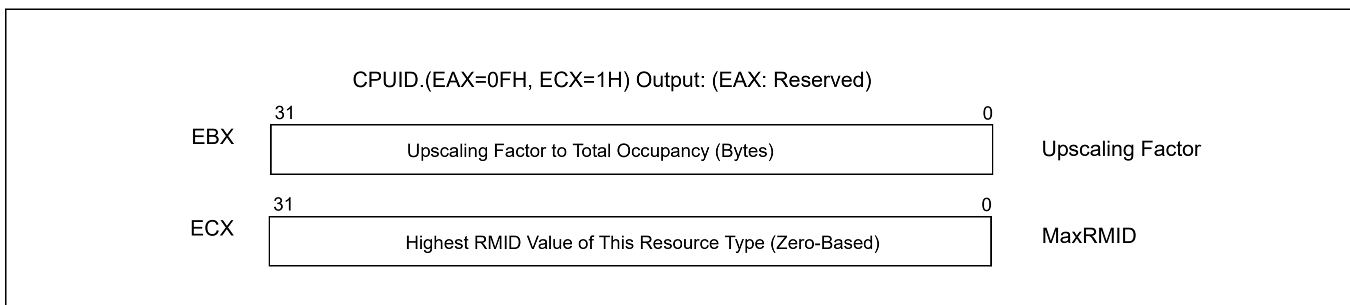


Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 17-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 17-22 and Table 17-18) that software must program with IA32_QM_EVTSEL in order to retrieve event data. After software configures IA32_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32_QM_CTR. The raw numerical value reported from IA32_QM_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 17-21.

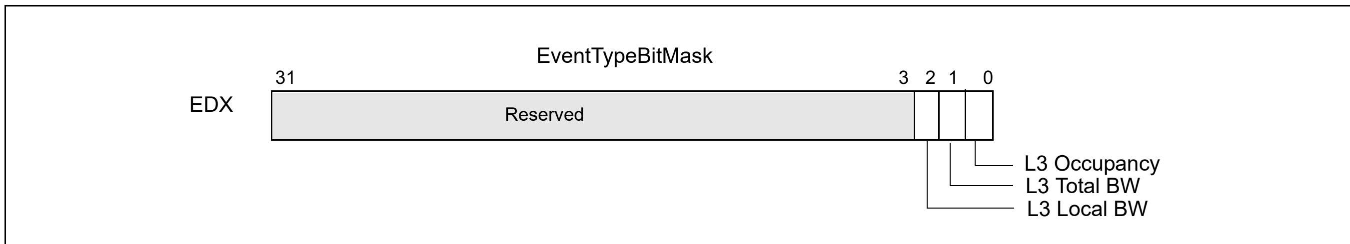


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))

17.17.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX would return with only bit 0 set. The corresponding event ID can be looked up from Table 17-18. The L3 occupancy data accumulated in IA32_QM_CTR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

17.17.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits will be set in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.
- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth (for instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction).

The corresponding Event ID can be looked up from Table 17-18. The L3 bandwidth data accumulated in IA32_QM_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.

Table 17-18. Monitoring Supported Event IDs

Event Type	Event ID	Context
L3 Cache Occupancy	01H	Cache Monitoring Technology
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM
Reserved	All other event codes	N/A

17.17.6 Monitoring Resource RMID Association

After Monitoring and sub-features has been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32_PQR_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32_PQR_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 17-23. Software specifies the active RMID to monitor in the IA32_PQR_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from Ceil (LOG_2 (1 + CPUID.(EAX=0FH, ECX=0):EBX[31:0])). The value of IA32_PQR_ASSOC after power-on is 0.

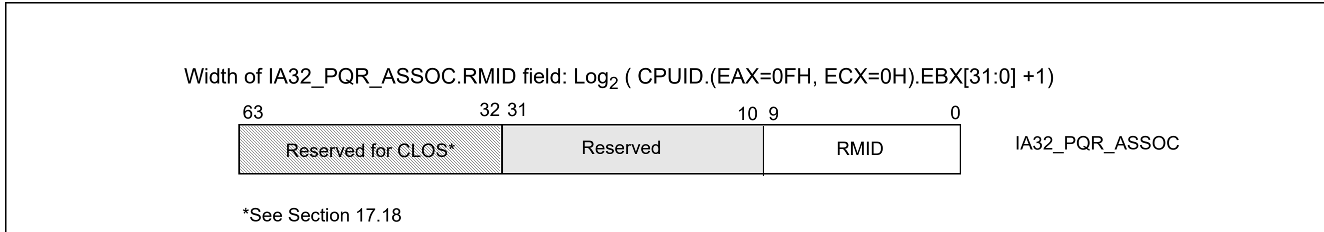


Figure 17-23. IA32_PQR_ASSOC MSR

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32_PQR_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g. RMID 31) may only be supported by one resource type but not by another resource type.

17.17.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported

back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g. no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32_QM_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 17-24. Bits IA32_QM_EVTSEL.EvtID (bits 7:0) specify an event code of a supported resource type for hardware to report monitored data associated with IA32_QM_EVTSEL.RMID (bits 41:32). Software can configure IA32_QM_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32_QM_EVTSEL.RMID matches that of IA32_PQR_ASSOC.RMID. Supported event codes for the IA32_QM_EVTSEL register are shown in Table 17-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

  Software can program an RMID / Event ID pair into the IA32_QM_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 17.17.5, which covers feature-specific details.

  Thread access to the IA32_QM_EVTSEL and IA32_QM_CTR MSR pair should be serialized to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32_QM_CTR.
- IA32_QM_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32_QM_EVTSEL, then IA32_QM_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32_QM_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32_QM_CTR.data (bits 61:0) should be ignored. Therefore, IA32_QM_CTR.data (bits 61:0) is valid only if bit 63 and 62 are both clear. For Cache Monitoring Technology, software can convert IA32_QM_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.

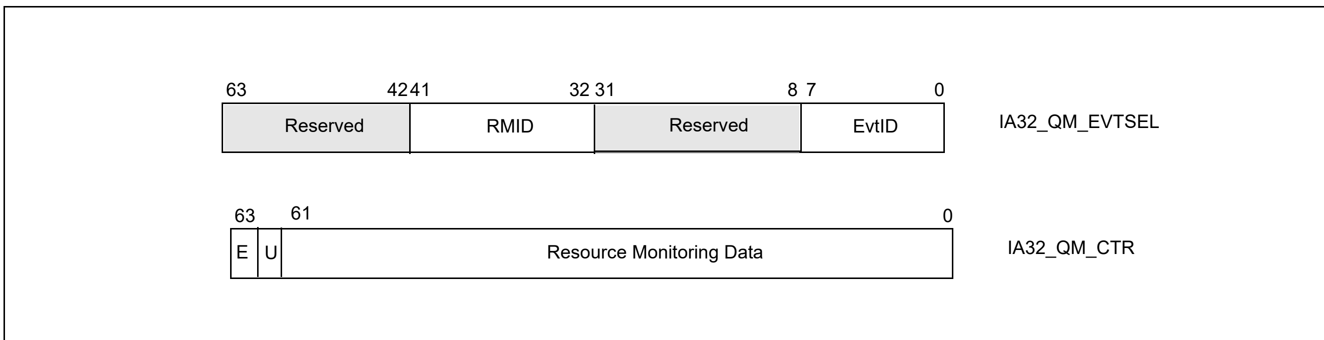


Figure 17-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs

17.17.8 Monitoring Programming Considerations

Figure 17-23 illustrates how system software can program IA32_QOSEVTSEL and IA32_QM_CTR to perform resource monitoring.

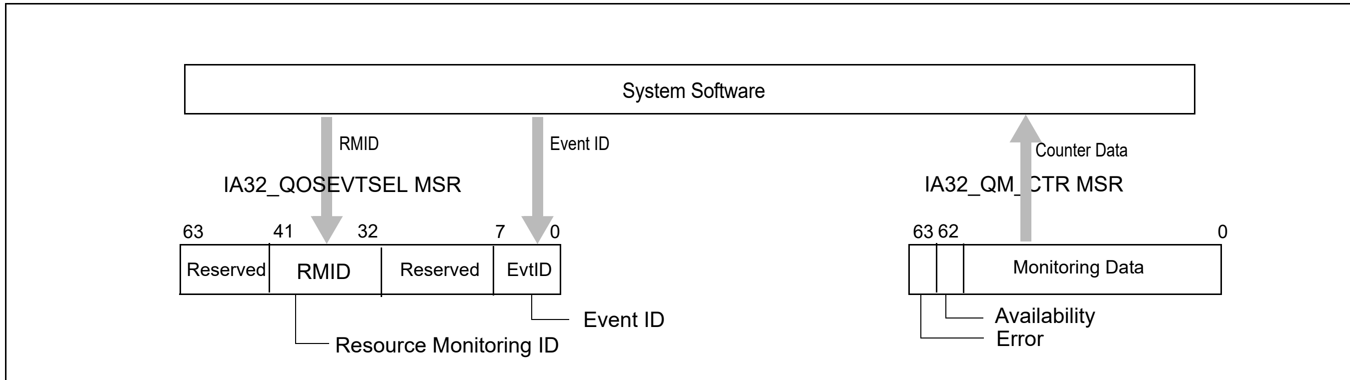


Figure 17-25. Software Usage of Cache Monitoring Resources

Though the field provided in IA32_QM_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used will be proportional to the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any future changes to counter width will be enumerated to software.

17.17.8.1 Monitoring Dynamic Configuration

Both the IA32_QM_EVTSEL and IA32_PQR_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- An RMID exceeding the maxRMID is used.

17.17.8.2 Monitoring Operation With Power Saving Features

Note that some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.

17.17.8.3 Monitoring Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler’s data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

17.17.8.4 Monitoring Operation with RAS Features

In general the Reliability, Availability and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses these may be tracked and may influence the shared resource monitoring counter values.

17.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and subset of communication-focused Intel Xeon processors E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Some future Intel platforms may also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter.

Future Intel processors introduce the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

17.18.1 Introduction to Cache Allocation Technology (CAT)

Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.

Software can determine which levels are supported in a give platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size,
- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks,
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs, and
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.

The Cache Allocation Technology feature enables more cache resources (i.e. cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 17-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSes, VMMs, containers and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e. OS/VMM) to support such resource control. Note that in Figure 17-26 the L3 Cache is shown as an example resource.

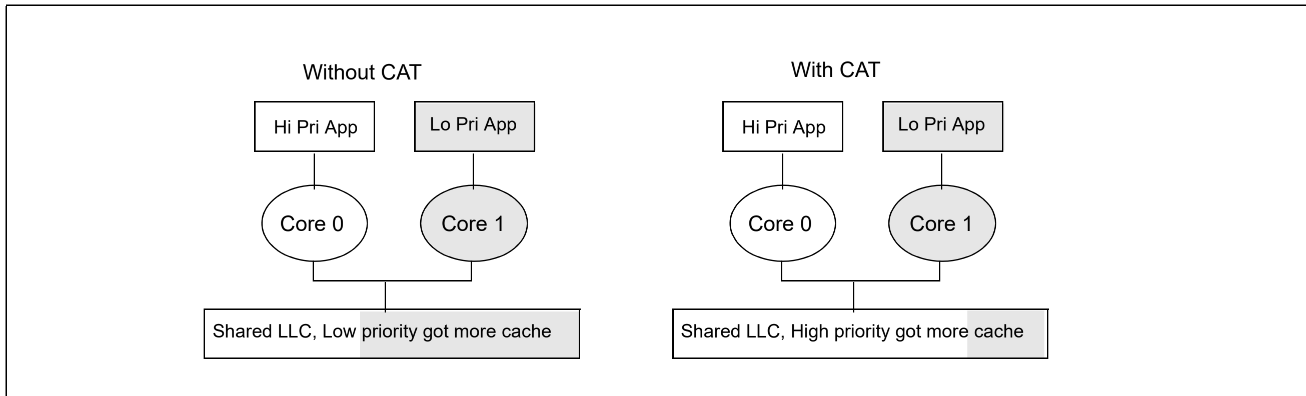


Figure 17-26. Cache Allocation Technology Allocates More Resource to High Priority Applications

17.18.2 Cache Allocation Technology Architecture

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS or CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32_PQR_ASSOC MSR or PQR) to allow the OS/VMM to specify a COS when an application, thread or VM is scheduled.

The usage of Classes of Service (COS) are consistent across resources - and a COS may have multiple re-source control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of COS tags per resource for instance, the COS management overhead is constant. Cache allocation for the indicated application/thread/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32_resourceType_MASK_n MSRs, where resourceType indicates a resource type (e.g. “L3” for the L3 cache) and n indicates a COS number.

The basic ingredients of Cache Allocation Technology are as follows:

- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled,
- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a COS to an executing software thread (i.e. associating the active CR3 of a logical processor with the COS in IA32_PQR_ASSOC),
- Implementation-dependent mechanisms to indicate which COS is associated with a memory access and to enforce the cache allocation on a per COS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bitlength of the capacity mask available generally depends on the configuration of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported COS may vary for each resource type, and these details can be enumerated via CPUID.

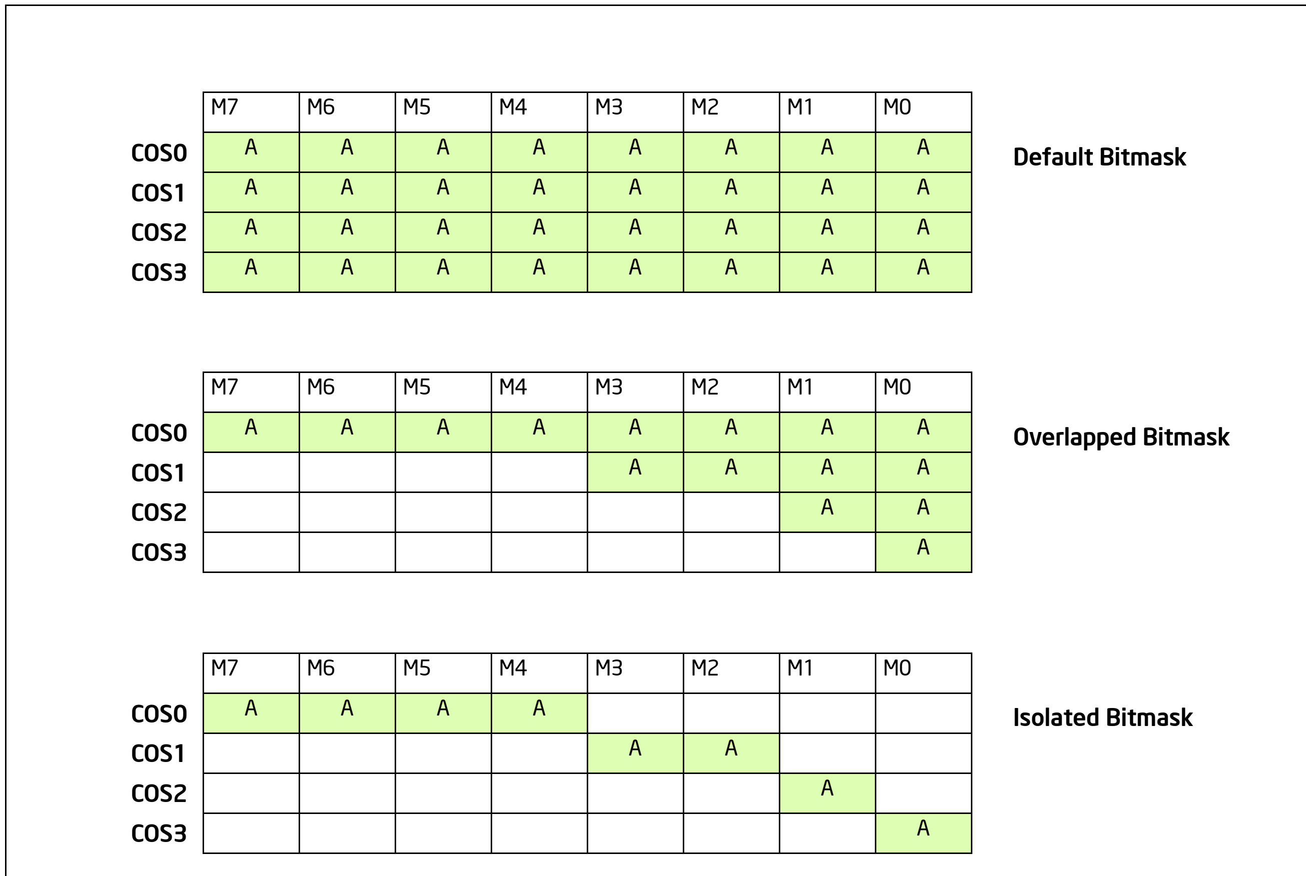


Figure 17-27. Examples of Cache Capacity Bitmasks

Sample cache capacity bitmasks for a bitlength of 8 are shown in Figure 17-27. Please note that all (and only) contiguous '1' combinations are allowed (e.g. FFFFH, 0FF0H, 003CH, etc.). Attempts to program a value without contiguous '1's (including zero) will result in a general protection fault (#GP(0)). It is generally expected that in way-based implementations, one capacity mask bit corresponds to some number of ways in cache, but the specific mapping is implementation-dependent. In all cases, a mask bit set to '1' specifies that a particular Class of Service can allocate into the cache subset represented by that bit. A value of '0' in a mask bit specifies that a Class of Service cannot allocate into the given cache subset. In general, allocating more cache to a given application is usually beneficial to its performance.

Figure 17-27 also shows three examples of sets of Cache Capacity Bitmasks. For simplicity these are represented as 8-bit vectors, though this may vary depending on the implementation and how the mask is mapped to the available cache capacity. The first example shows the default case where all 4 Classes of Service (the total number of COS are implementation-dependent) have full access to the cache. The second case shows an overlapped case, which would allow some lower-priority threads share cache space with the highest priority threads. The third case shows various non-overlapped partitioning schemes. As a matter of software policy for extensibility COS0 should typically be considered and configured as the highest priority COS, followed by COS1, and so on, though there is no hardware restriction enforcing this mapping. When the system boots all threads are initialized to COS0, which has full access to the cache by default.

Though the representation of the CBMs looks similar to a way-based mapping they are independent of any specific enforcement implementation (e.g. way partitioning.) Rather, this is a convenient manner to represent capacity, overlap and isolation of cache space. For example, executing a POPCNT instruction (population count of set bits) on the capacity bitmask can provide the fraction of cache space that a class of service can allocate into. In addition to the fraction, the exact location of the bits also shows whether the class of service overlaps with other classes of service or is entirely isolated in terms of cache space used.

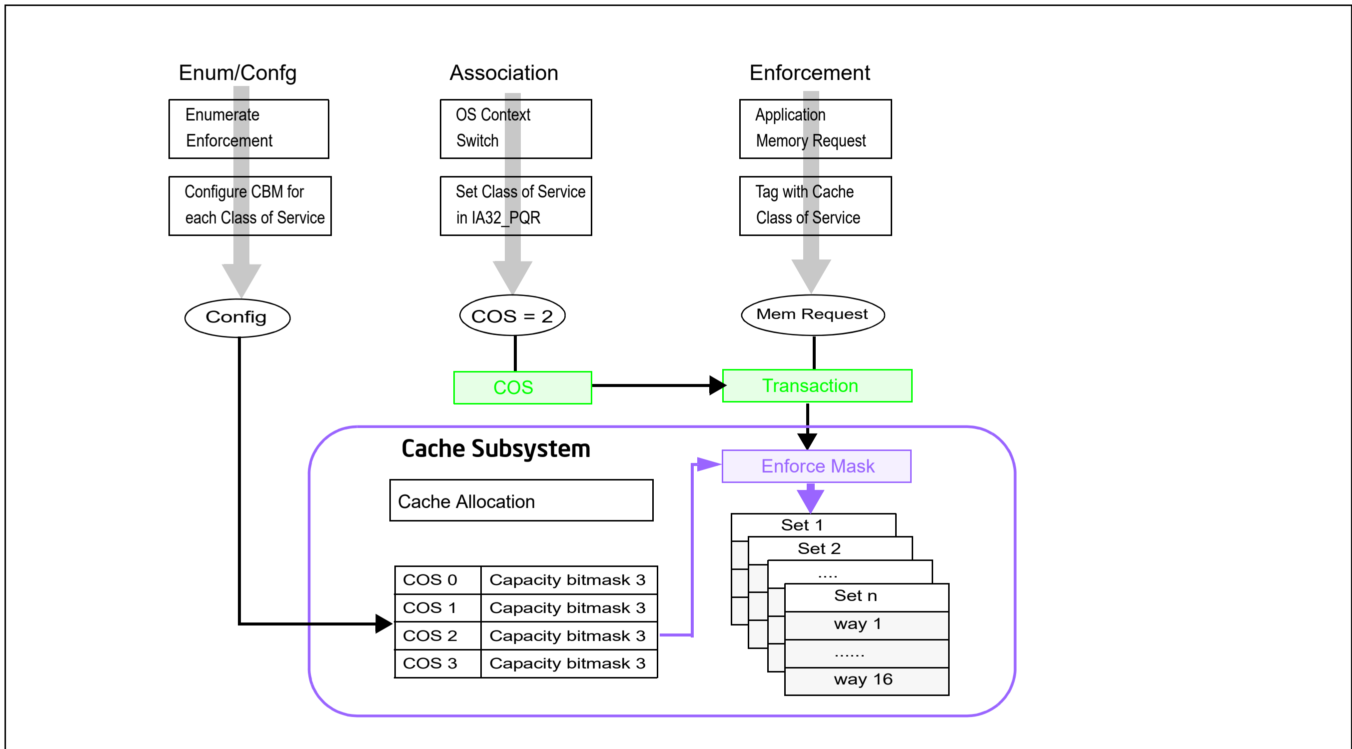


Figure 17-28. Class of Service and Cache Capacity Bitmasks

Figure 17-28 shows how the Cache Capacity Bitmasks and the per-logical-processor Class of Service are logically used to enable Cache Allocation Technology. All (and only) contiguous 1's in the CBM are permitted. The length of CBM may vary from resource to resource or between processor generations and can be enumerated using CPUID. From the available mask set and based on the goals of the OS/VMM (shared or isolated cache, etc.) bitmasks are selected and associated with different classes of service. For the available Classes of Service the associated CBMs can be programmed via the global set of CAT configuration registers (in the case of L3 CAT, via the IA32_L3_MASK_n MSRs, where "n" is the Class of Service, starting from zero). In all architectural implementations supporting CPUID it is possible to change the CBMs dynamically, during program execution, unless stated otherwise by Intel.

The currently running application's Class of Service is communicated to the hardware through the per-logical-processor PQR MSR (IA32_PQR_ASSOC MSR). When the OS schedules an application thread on a logical processor, the application thread is associated with a specific COS (i.e. the corresponding COS in the PQR) and all requests to the CAT-capable resource from that logical processor are tagged with that COS (in other words, the application thread is configured to belong to a specific COS). The cache subsystem uses this tagged request information to enforce QoS. The capacity bitmask may be mapped into a way bitmask (or a similar enforcement entity based on the implementation) at the cache before it is applied to the allocation policy. For example, the capacity bitmask can be an 8-bit mask and the enforcement may be accomplished using a 16-way bitmask for a cache enforcement implementation based on way partitioning.

The following sections describe extensions of CAT such as Code and Data Prioritization (CDP), followed by details on specific features such as L3 CAT, L3 CDP, and L2 CAT. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to detect the set of supported features.

17.18.3 Code and Data Prioritization (CDP) Technology

Code and Data Prioritization Technology is an extension of CAT. CDP enables isolation and separate prioritization of code and data fetches to the L3 cache in a software configurable manner, which can enable workload prioritization

and tuning of cache capacity to the characteristics of the workload. CDP extends Cache Allocation Technology (CAT) by providing separate code and data masks per Class of Service (COS).

By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled,

- the CAT mask MSRs are re-mapped into interleaved pairs of mask MSRs for data or code fetches (see Figure 17-29),
- the range of COS for CAT is re-indexed, with the lower-half of the COS range available for CDP.

Using the CDP feature, virtual isolation between code and data can be configured on the L3 cache if desired, similar to how some processor cache levels provide separate L1 data and L1 instruction caches.

Like the CAT feature, CDP may be dynamically configured by privileged software at any point during normal system operation, including dynamically enabling or disabling the feature provided that certain software configuration requirements are met (see Section 17.18.5).

An example of the operating mode of CDP is shown in Figure 17-29. Shown at the top are traditional CAT usage models where capacity masks map 1:1 with a COS number to enable control over the cache space which a given COS (and thus applications, threads or VMs) may occupy. Shown at the bottom are example mask configurations where CDP is enabled, and each COS number maps 1:2 to two masks, one for code and one for data. This enables code and data to be either overlapped or isolated to varying degrees either globally or on a per-COS basis, depending on application and system needs.

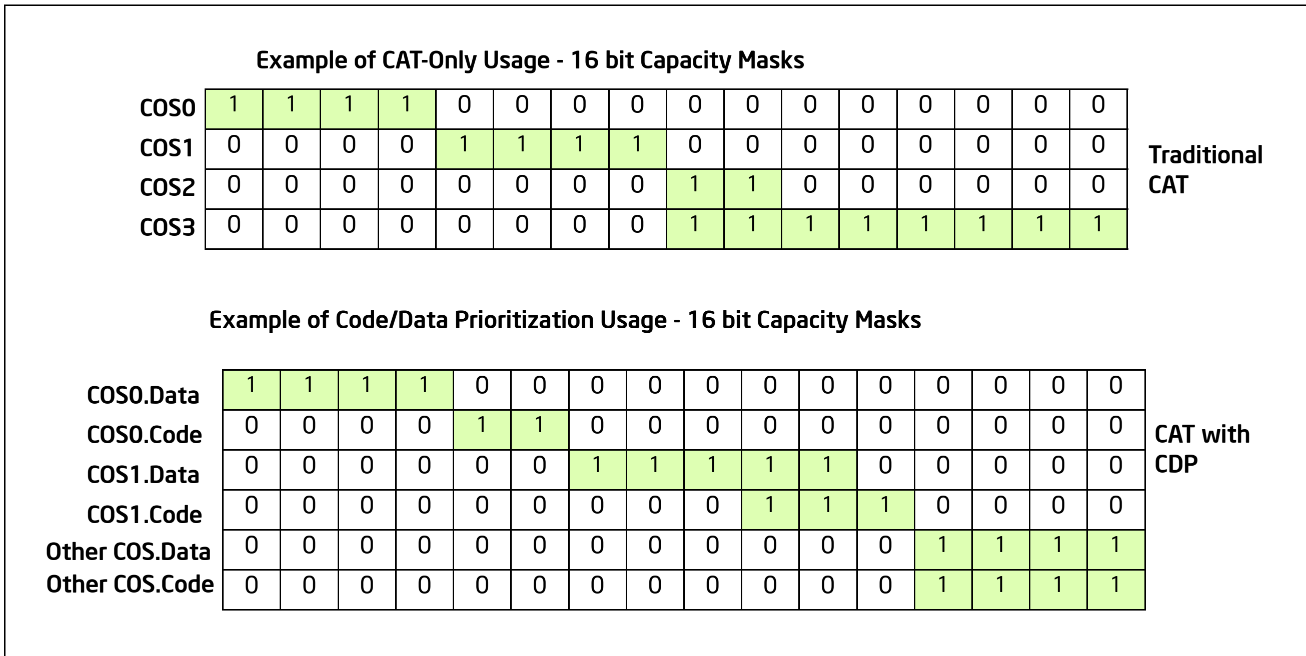


Figure 17-29. Code and Data Capacity Bitmasks of CDP

When CDP is enabled, the existing mask space for CAT-only operation is split. As an example if the system supports 16 CAT-only COS, when CDP is enabled the same MSR interfaces are used, however half of the masks correspond to code, half correspond to data, and the effective number of COS is reduced by half. Code/Data masks are defined per-COS and interleaved in the MSR space as described in subsequent sections.

In cases where CPUID exposes a non-even number of supported Classes of Service for the CAT or CDP features, software using CDP should use the lower matched pairs of code/data masks, and any upper unpaired masks should not be used. As an example, if CPUID exposes 5 CLOS, when CDP is enabled then two code/data pairs are available (masks 0/1 for CLOS[0] data/code and masks 2/3 for CLOS[1] data/code), however the upper un-paired mask should not be used (mask 4 in this case).

17.18.4 Enabling Cache Allocation Technology Usage Flow

Figure 17-30 illustrates the key steps for OS/VMM to detect support of Cache Allocation Technology and enable priority-based resource allocation for a CAT-capable resource.

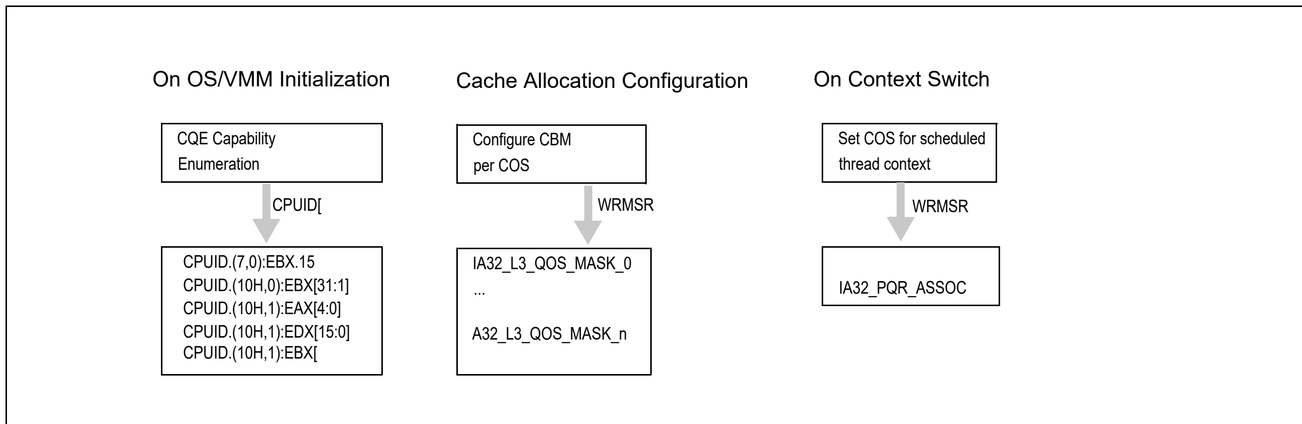


Figure 17-30. Cache Allocation Technology Usage Flow

Enumeration and configuration of L2 CAT is similar to L3 CAT, however CPUID details and MSR addresses differ. Common CLOS are used across the features.

17.18.4.1 Enumeration and Detection Support of Cache Allocation Technology

Software can query processor support of CAT capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software must use CPUID leaf 10H to enumerate additional details of available resource types, classes of services and capability bitmasks. The programming interfaces provided by Cache Allocation Technology include:

- CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) and its sub-functions provide information on available resource types, and CAT capability for each resource type (see Section 17.18.4.2).
- IA32_L3_MASK_n: A range of MSRs is provided for each resource type, each MSR within that range specifying a software-configured capacity bitmask for each class of service. For L3 with Cache Allocation support, the CBM is specified using one of the IA32_L3_QOS_MASK_n MSR, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive. See Section 17.18.4.3 for details.
- IA32_L2_MASK_n: A range of MSRs is provided for L2 Cache Allocation Technology, enabling software control over the amount of L2 cache available for each CLOS. Similar to L3 CAT, a CBM is specified for each CLOS using the set of registers, IA32_L2_QOS_MASK_n MSR, where 'n' ranges from zero to the maximum CLOS number reported for L2 CAT in CPUID. See Section 17.18.4.3 for details.

  The L2 mask MSRs are scoped at the same level as the L2 cache (similarly, the L3 mask MSRs are scoped at the same level as the L3 cache). Software may determine which logical processors share an MSR (for instance local to a core, or shared across multiple cores) by performing a write to one of these MSRs and noting which logical threads observe the change. Example flows for a similar method to determine register scope are described in Section 15.5.2, "System Software Recommendation for Managing CMCI and Machine Check Resources". Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.
- IA32_PQR_ASSOC.CLOS: The IA32_PQR_ASSOC MSR provides a COS field that OS/VMM can use to assign a logical processor to an available COS. The set of COS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software COS management overhead at context swap time. See Section 17.18.4.4 for details.

17.18.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) provides two or more sub-functions:

- CAT Enumeration leaf sub-function 0 enumerates available resource types that support allocation control, i.e. by executing CPUID with EAX=10H and ECX=0H. Each supported resource type is represented by a bit field in CPUID.(EAX=10H, ECX=0):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of that resource type (see Figure 17-31).

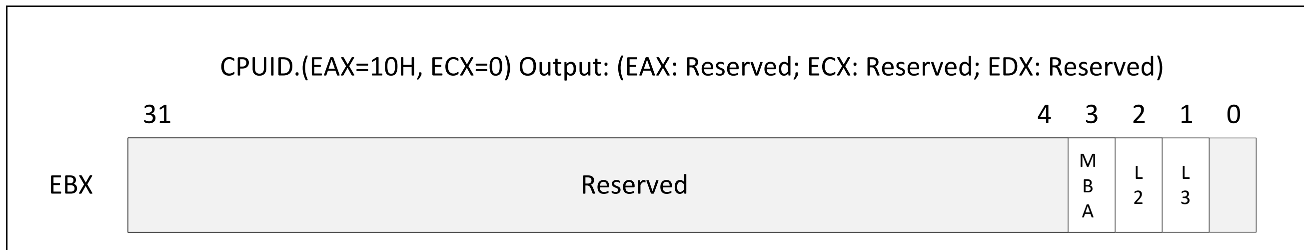


Figure 17-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

  — For ECX>0, EAX[4:0] reports the length of the capacity bitmask length (ECX=1 or 2 for L2 CAT or L3 CAT respectively) using minus-one notation, e.g., a value of 15 corresponds to the capacity bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

- Sub-functions of CPUID.EAX=10H with a non-zero ECX input matching a supported ResID enumerate the specific enforcement details of the corresponding ResID. The capabilities enumerated include the length of the capacity bitmasks and the number of Classes of Service for a given ResID. Software should query the capability of each available ResID that supports CAT from a sub-leaf of leaf 10H using the sub-leaf index reported by the corresponding non-zero bit in CPUID.(EAX=10H, ECX=0):EBX[31:1] in order to obtain additional feature details.
- CAT capability for L3 is enumerated by CPUID.(EAX=10H, ECX=1H), see Figure 17-32. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=1) are:

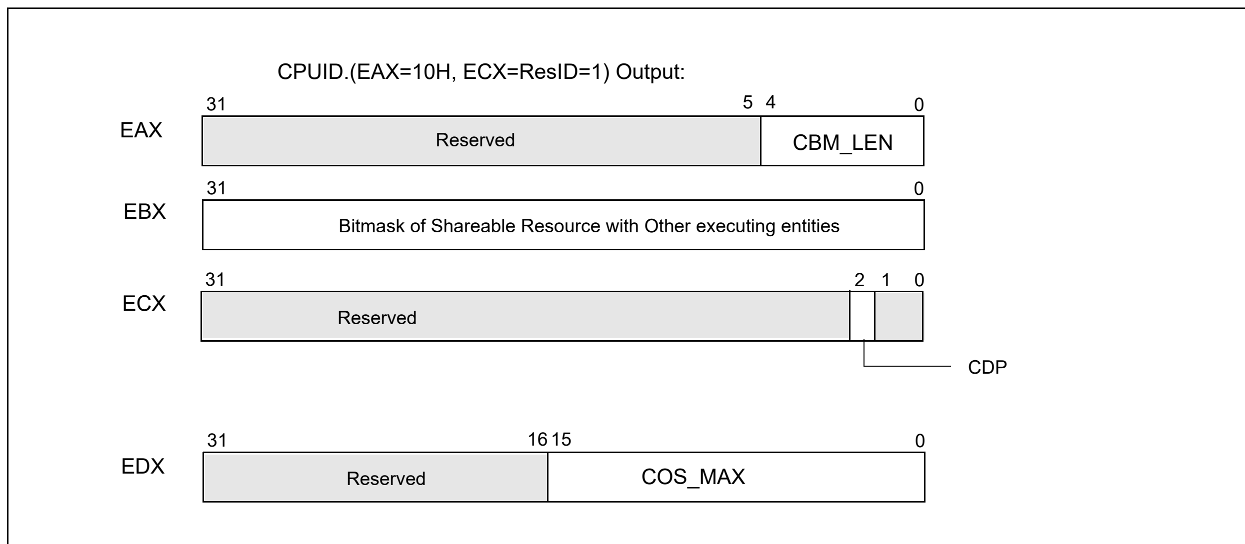


Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration

— CPUID.(EAX=10H, ECX=ResID=1):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

— CPUID.(EAX=10H, ECX=1):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L3 allocation may be used by other entities in the platform (e.g. an integrated graphics engine or hardware units outside the processor core and have direct access to L3). Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

— CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2]: If 1, indicates Code and Data Prioritization Technology is supported (see Section 17.18.5). Other bits of CPUID.(EAX=10H, ECX=1):ECX are reserved.

— CPUID.(EAX=10H, ECX=1):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

- CAT capability for L2 is enumerated by CPUID.(EAX=10H, ECX=2H), see Figure 17-33. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=2) are:

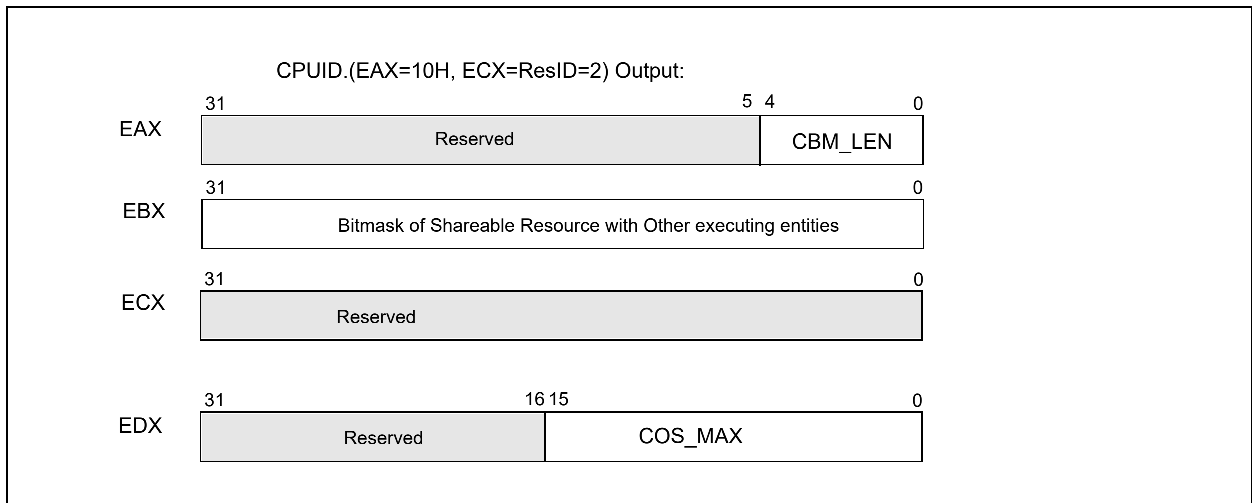


Figure 17-33. L2 Cache Allocation Technology

— CPUID.(EAX=10H, ECX=ResID=2):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

— CPUID.(EAX=10H, ECX=2):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L2 allocation may be used by other entities in the platform. Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

— CPUID.(EAX=10H, ECX=2):ECX: reserved.

— CPUID.(EAX=10H, ECX=2):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

A note on migration of Classes of Service (COS): Software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the performance of the Cache Allocation Technology feature may result if COS are migrated frequently. This is aligned with the industry-standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor

caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

17.18.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of COS supported using CPUID (see Section 17.18.4.2), each COS needs to be programmed with a CBM to dictate its available cache via a write to the corresponding IA32_resourceType_MASK_n register, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive, and 'resourceType' corresponds to a specific resource as enumerated by the set bits of CPUID.(EAX=10H, ECX=0):EAX[31:1], for instance, ‘L2’ or ‘L3’ cache.

A hierarchy of MSRs is reserved for Cache Allocation Technology registers of the form IA32_resourceType_MASK_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resourceType is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

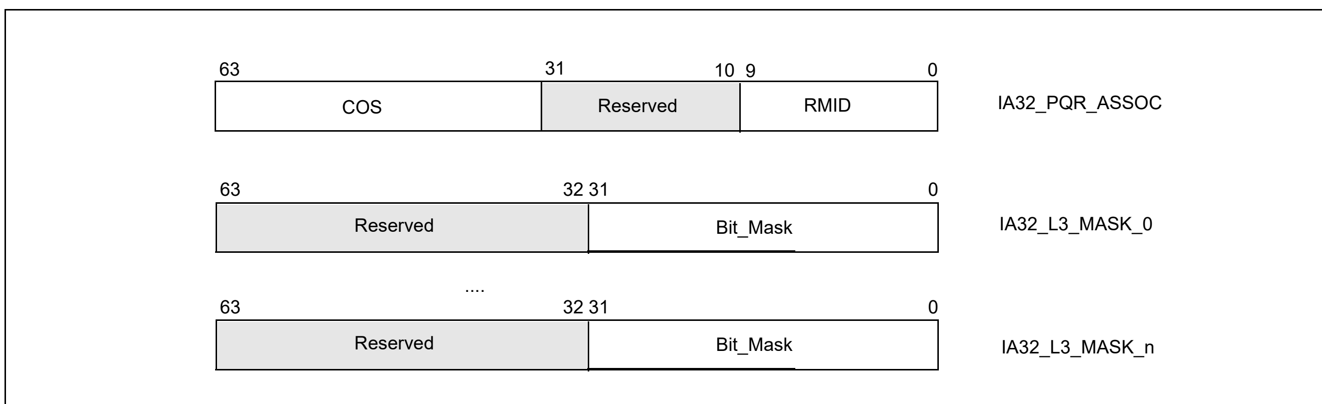


Figure 17-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32_L2_MASK_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 17-34 and Figure 17-35 provide an overview of the relevant registers.

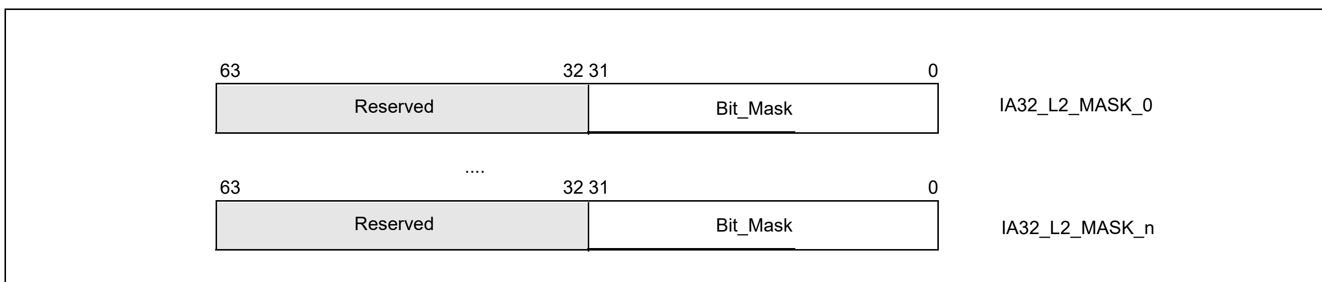


Figure 17-35. IA32_L2_MASK_n MSRs

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (COS) using the IA32_PQR_ASSOC MSR as described in Chapter 17, “Class of Service to Cache Mask Association: Common Across Allocation Features”.

17.18.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features

After configuring the available classes of service with the preferred set of capacity bitmasks, the OS/VMM can set the IA32_PQR_ASSOC.COS of a logical processor to the class of service with the desired CBM when a thread context switch occurs. This allows the OS/VMM to indicate which class of service an executing thread/VM belongs within. Each logical processor contains an instance of the IA32_PQR_ASSOC register at MSR location 0C8FH, and Figure 17-34 shows the bit field layout for this register. Bits[63:32] contain the COS field for each logical processor.

Note that placing the RMID field within the same PQR register enables both RMID and CLOS to be swapped at context swap time for simultaneous use of monitoring and allocation features with a single register write for efficiency.

When CDP is enabled, Specifying a COS value in IA32_PQR_ASSOC.COS greater than MAX_COS_CDP =(CPUID.(EAX=10H, ECX=1):EDX[15:0] >> 1) will cause undefined performance impact to code and data fetches.

Note that if the IA32_PQR_ASSOC.COS is never written then the CAT capability defaults to using COS 0, which in turn is set to the default mask in IA32_L3_MASK_0 - which is all "1"s (on reset). This essentially disables the enforcement feature by default or for legacy operating systems and software.

See Section 17.18.6, "Cache Allocation Technology Programming Considerations" for important COS programming considerations including maximum values when using CAT and CDP.

17.18.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

CDP is an extension of CAT. The presence of the CDP feature is enumerated via CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] (see Figure 17-32). Most of the CPUID.(EAX=10H, ECX=1) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, COS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] =1, the processor supports CDP and provides a new MSR IA32_L3_QOS_CFG at address 0C81H. The layout of IA32_L3_QOS_CFG is shown in Figure 17-36. The bit field definition of IA32_L3_QOS_CFG are:

- Bit 0: L3 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.COS is COS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

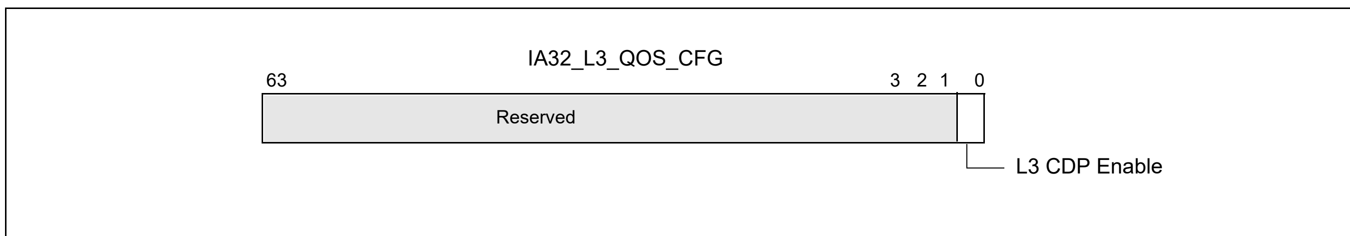


Figure 17-36. Layout of IA32_L3_QOS_CFG

IA32_L3_QOS_CFG default values are all 0s at RESET, the mask MSRs are all 1s. Hence. all logical processors are initialized in COS0 allocated with the entire L3 with CDP disabled, until software programs CAT and CDP.

Before enabling or disabling CDP, software should write all 1's to all of the CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs). When enabling CDP, software should also ensure that only COS number which are valid in CDP operation is used, otherwise undefined behavior may result. For instance in a case with 16 CAT COS, since COS are reduced by half when CDP is enabled, software should ensure that only COS 0-7 are in use before enabling CDP (along with writing 1's to all mask bits before enabling or disabling CDP).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service

when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

17.18.5.1 Mapping Between L3 CDP Masks and CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. The re-mapping is shown in Table 17-19.

Table 17-19. Re-indexing of COS Numbers and Mapping to CAT/CDP Mask MSRs

Mask MSR	CAT-only Operation	CDP Operation
IA32_L3_QOS_Mask_0	COS0	COS0.Data
IA32_L3_QOS_Mask_1	COS1	COS0.Code
IA32_L3_QOS_Mask_2	COS2	COS1.Data
IA32_L3_QOS_Mask_3	COS3	COS1.Code
IA32_L3_QOS_Mask_4	COS4	COS2.Data
IA32_L3_QOS_Mask_5	COS5	COS2.Code
....	....	....
IA32_L3_QOS_Mask_'2n'	COS'2n'	COS'n'.Data
IA32_L3_QOS_Mask_'2n+1'	COS'2n+1'	COS'n'.Code

One can derive the MSR address for the data mask or code mask for a given COS number 'n' by:

- data_mask_address (n) = base + (n <<1), where base is the address of IA32_L3_QOS_MASK_0.
- code_mask_address (n) = base + (n <<1) +1.

When CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over data placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 17-29).

Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003 and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

17.18.5.2 L3 CAT: Disabling CDP

Before enabling or disabling CDP, software should write all 1's to all of the CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

17.18.6 Cache Allocation Technology Programming Considerations

17.18.6.1 Cache Allocation Technology Dynamic Configuration

Both the CAT masks and CQM registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,

- Accessing a QOS mask register outside the supported COS (the max COS number is specified in CPUID.(EAX=10H, ECX=ResID):EDX[15:0]), or
- Writing a COS greater than the supported maximum (specified as the maximum value of CPUID.(EAX=10H, ECX=ResID):EDX[15:0] for all valid ResID values) is written to the IA32_PQR_ASSOC.CLOS field.

When CDP is enabled, specifying a COS value in IA32_PQR_ASSOC.COS outside of the lower half of the COS space will cause undefined performance impact to code and data fetches due to MSR space re-indexing into code/data masks when CDP is enabled.

When reading the IA32_PQR_ASSOC register the currently programmed COS on the core will be returned.

When reading an IA32_*resourceType*_MASK_*n* register the current capacity bit mask for COS 'n' will be returned.

As noted previously, software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the accuracy of the Cache Allocation feature may result if COS are migrated frequently. This is aligned with the industry standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

17.18.6.2 Cache Allocation Technology Operation With Power Saving Features

Note that the Cache Allocation Technology feature cannot be used to enforce cache coherency, and that some advanced power management features such as C-states which may shrink or power off various caches within the system may interfere with CAT hints - in such cases the CAT bitmasks are ignored and the other features take precedence. If the highest possible level of CAT differentiation or determinism is required, disable any power-saving features which shrink the caches or power off caches. The details of the power management interfaces are typically implementation-specific, but can be found at *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

If software requires differentiation between threads but not absolute determinism then in many cases it is possible to leave power-saving cache shrink features enabled, which can provide substantial power savings and increase battery life in mobile platforms. In such cases when the caches are powered off (e.g., package C-states) the entire cache of a portion thereof may be powered off. Upon resuming an active state any new incoming data to the cache will be filled subject to the cache capacity bitmasks. Any data in the cache prior to the cache shrink or power off may have been flushed to memory during the process of entering the idle state, however, and is not guaranteed to remain in the cache. If differentiation between threads is the goal of system software then this model allows substantial power savings while continuing to deliver performance differentiation. If system software needs optimal determinism then power saving modes which flush portions of the caches and power them off should be disabled.

NOTE

IA32_PQR_ASSOC is saved and restored across C6 entry/exit. Similarly, the mask register contents are saved across package C-state entry/exit and are not lost.

17.18.6.3 Cache Allocation Technology Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and mask registers are unmodified across an SMI delivery. Thus, the execution of SMM handler code can interact with the Cache Allocation Technology resource and manifest some degree of non-determinism to the non-SMM software stack. An SMM handler may also perform certain system-level or power management practices that affect CAT operation.

It is possible for an SMM handler to minimize the impact on data determinism in the cache by reserving a COS with a dedicated partition in the cache. Such an SMM handler can switch to the dedicated COS immediately upon entering SMM, and switching back to the previously running COS upon exit.

17.18.6.4 Associating Threads with CAT/CDP Classes of Service

Threads are associated with Classes of Service (CLOS) via the per-logical-processor IA32_PQR_ASSOC MSR. The same COS concept applies to both CAT and CDP (for instance, COS[5] means the same thing whether CAT or CDP is in use, and the COS has associated resource usage constraint attributes including cache capacity masks). The mapping of COS to mask MSRs does change when CDP is enabled, according to the following guidelines:

- In CAT-only Mode - one set of bitmasks in one mask MSR control both code and data.
  - Each COS number map 1:1 with a capacity mask on the applicable resource (e.g., L3 cache).
- When CDP is enabled,
  - Two mask sets exist for each COS number, one for code, one for data.
  - Masks for code/data are interleaved in the MSR address space (see Table 17-19).

17.18.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core, and is introduced on future Intel processors. This feature provides a method to control applications which may be over-utilizing bandwidth relative to their priority in environments such as the data-center.

The MBA feature uses existing constructs from the Resource Director Technology (RDT) feature set including Classes of Service (CLOS). A given CLOS used for L3 CAT for instance means the same thing as a CLOS used for MBA. Infrastructure such as the MSR used to associate a thread with a CLOS (the IA32_PQR_ASSOC_MSR) and some elements of the CPUID enumeration (such as CPUID leaf 10H) are shared.

- The high-level implementation of Memory Bandwidth Allocation is shown in Figure 17-37.

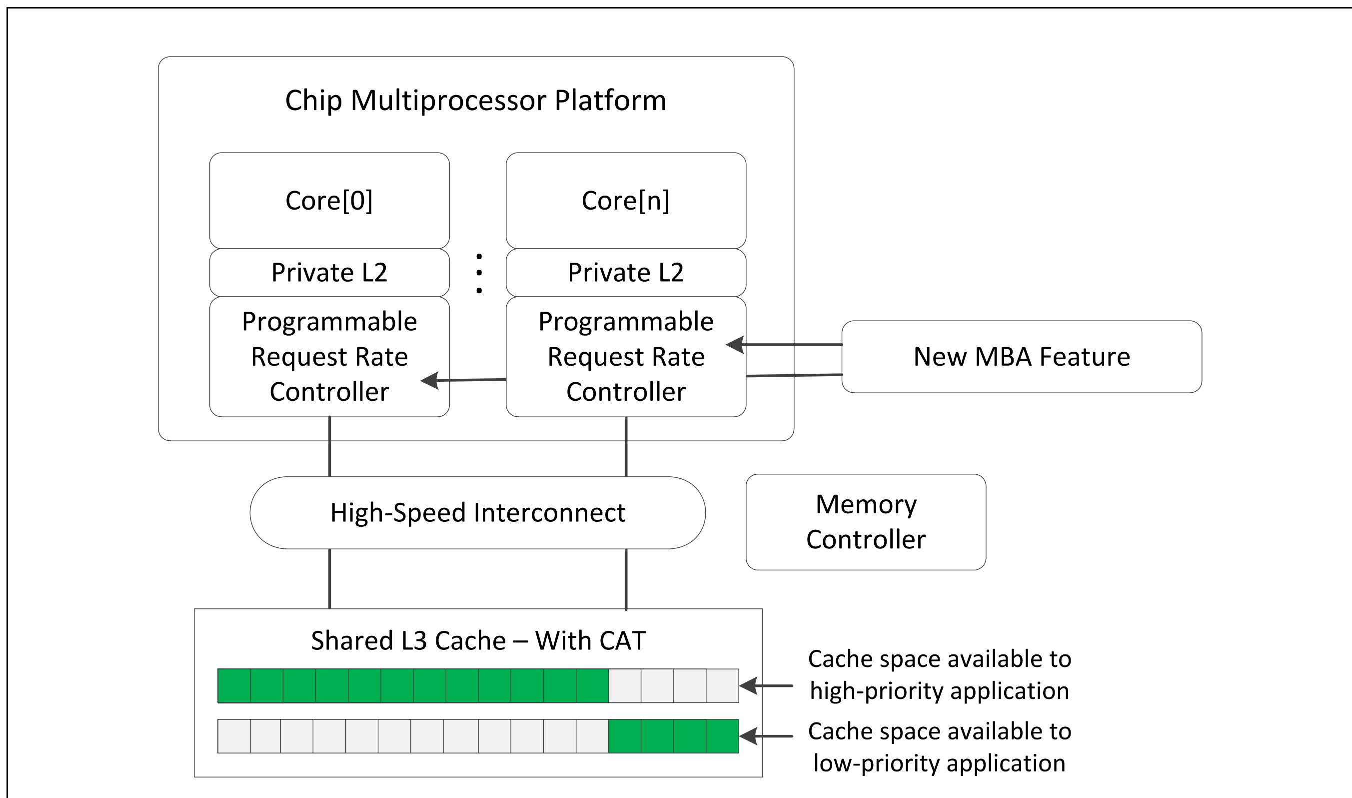


Figure 17-37. A High-Level Overview of the MBA Feature

As shown in Figure 17-37 the MBA feature introduces a programmable request rate controller between the cores and the high-speed interconnect, enabling indirect control over memory bandwidth for cores over-utilizing bandwidth relative to their priority. For instance, high-priority cores may be run un-throttled, but lower priority cores generating an excessive amount of traffic may be throttled to enable more bandwidth availability for the high-priority cores.

Since MBA uses a programmable rate controller between the cores and the interconnect, higher-level shared caches and memory controller, bandwidth to these caches may also be reduced, so care should be taken to throttle only bandwidth-intense applications which do not use the off-core caches effectively.

The throttling values exposed by MBA are approximate, and are calibrated to specific traffic patterns. As work-load characteristics vary, the throttling values provided may affect each workload differently. In cases where precise control is needed, the Memory Bandwidth Monitoring (MBM) feature can be used as input to a software controller which makes decisions about the MBA throttling level to apply.

Enumeration and configuration details are discussed below followed by usage model considerations.

17.18.7.1 Memory Bandwidth Allocation Enumeration

Similar to other RDT features, enumeration of the presence and details of the MBA feature is provided via a sub-leaf of the CPUID instruction.

Key components of the enumeration are as follows.

- Support for the MBA feature on the processor, and if MBA is supported, the following details:
  - Number of supported Classes of Service (CLOS) for the processor.
  - The maximum MBA delay value supported (which also implicitly provides a definition of the granularity).
  - An indication of whether the delay values which can be programmed are linearly spaced or not.

The presence of any of the RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software may then use CPUID leaf 10H to enumerate additional details on the specific controls provided.

Through CPUID leaf 10H software may determine whether MBA is supported on the platform. Specifically, as shown in Figure 17-31, bit 3 of the EBX register indicates whether MBA is supported on the processor, and the bit position (3) constitutes a Resource ID (ResID) which allows enumeration of MBA details. For instance, if bit 3 is supported this implies the presence of CPUID.10H.[ResID=3] as shown in Figure 17-38 which provides the following details.

- CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] reports the maximum MBA throttling value supported, minus one. For instance, a value of 89 indicates that a maximum throttling value of 90 is supported. Additionally, in cases where a linear interface (see below) is supported then one hundred minus the maximum throttling value indicates the granularity, 10% in this example.
- CPUID.(EAX=10H, ECX=ResID=3):EBX is reserved.
- CPUID.(EAX=10H, ECX=ResID=3):ECX[2] reports whether the response of the delay values is linear (see text).
- CPUID.(EAX=10H, ECX=ResID=3):EDX[15:0] reports the number of Classes of Service (CLOS) supported for the feature (minus one). For instance, a reported value of 15 implies a maximum of 16 supported MBA CLOS.

The number of CLOS supported for the MBA feature may or may not align with other resources such as L3 CAT. In cases where the RDT features support different numbers of CLOS the lowest numerical CLOS support the common set of features, while higher CLOS may support a subset. For instance, if L3 CAT supports 8 CLOS while MBA supports 4 CLOS, all 8 CLOS would have L3 CAT masks available for cache control, but the upper 4 CLOS would not offer MBA support. In this case the upper 4 CLOS would not be subject to any throttling control. Software can manage supported resources / CLOS in order to either have consistent capabilities across CLOS by using the common subset or enable more flexibility by selectively applying resource control where needed based on careful CLOS and thread mapping. In all cases, CLOS[0] supports all RDT resource control features present on the platform.

Discussion on the interpretation and usage of the MBA delay values is provided in Section 17.18.7.2 on MBA configuration.

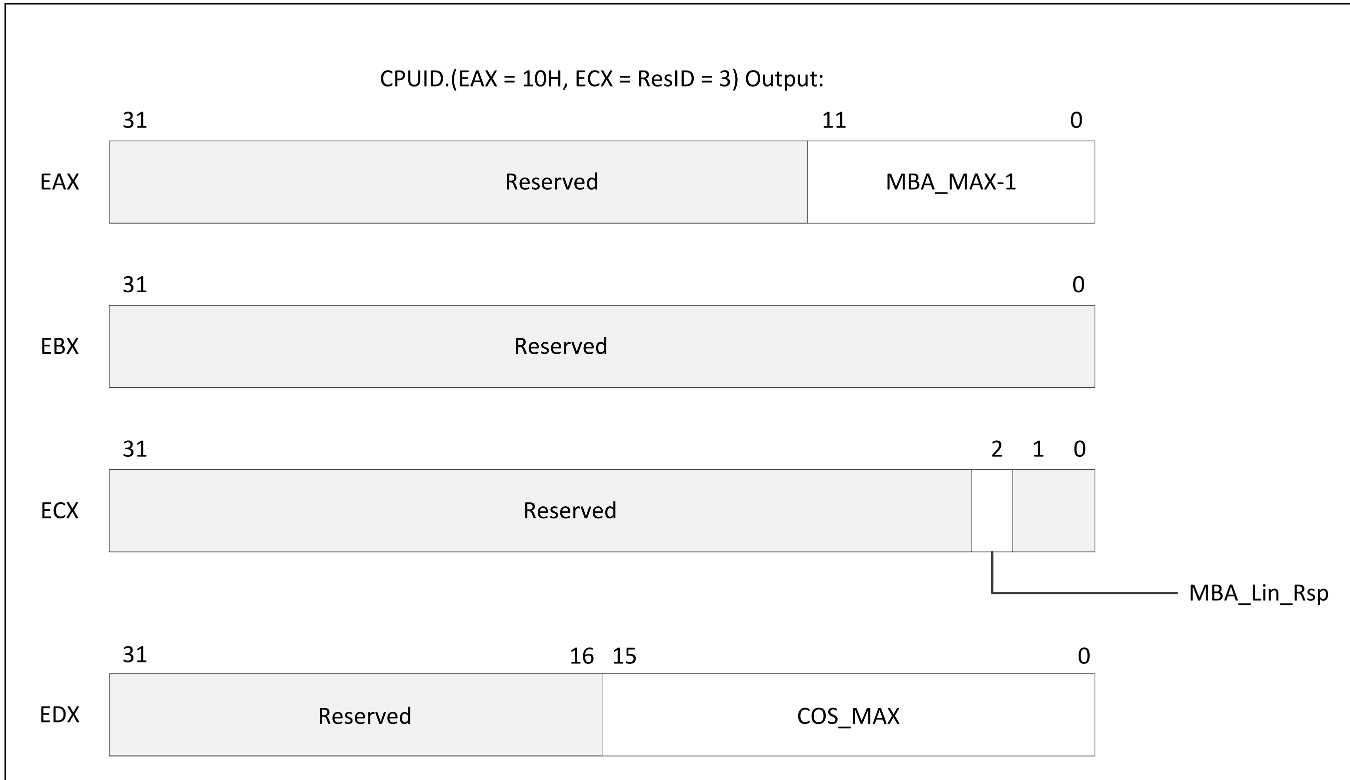


Figure 17-38. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification

17.18.7.2 Memory Bandwidth Allocation Configuration

The configuration of MBA takes consists of two processes once enumeration is complete.

- Association of threads to Classes of Service (CLOS) - accomplished in a common fashion across RDT features as described in Section 17.18.7.1 via the IA32_PQR_ASSOC MSR. As with features such as L3 CAT, software may update the CLOS field of the PQR MSR at context swap time in order to maintain the proper association of software threads to Classes of Service on the hardware. While logical processors may each be associated with independent CLOS, see Section 17.18.7.3 for important usage model considerations (initial versions of the MBA feature select the maximum delay value across threads).
- Configuration of the per-CLOS delay values, accomplished via the IA32_L2_QoS_Ext_BW_Thrtl_n MSR set shown in Table 17-20.

The MBA delay values which may be programmed range from zero (implying zero delay, and full bandwidth available) to the maximum (MBA_MAX) specified in CPUID as discussed in Section 17.18.7.1.

Software may select an MBA delay value then write the value into one or more of the IA32_L2_QoS_Ext_BW_Thrtl_n MSRs to update the delay values applied for a specific CLOS. As shown in Table 17.20 the base address of the MSRs is at D50H, and the range corresponds to the maximum supported CLOS from CPUID.(EAX=10H, ECX=ResID=1):EDX[15:0] as described in Section 17.18.7.1. For instance, if 16 CLOS are supported then the valid MSR range will extend from D50H through D5F inclusive.

Table 17-20. MBA Delay Value MSRs

Delay Value MSR	Address
IA32_L2_QoS_Ext_BW_Thrtl_0	D50H
IA32_L2_QoS_Ext_BW_Thrtl_1	D51H
IA32_L2_QoS_Ext_BW_Thrtl_2	D52H
….	….
IA32_L2_QoS_Ext_BW_Thrtl_'COS_MAX'	D50H + COS_MAX from CPUID.10H.3

The definition for the MBA delay value MSRs is provided in Figure 17.39. The lower 16 bits are used for MBA delay values, and values from zero to the maximum from the CPUID MBA_MAX-1 value are supported. Values outside this range will generate #GP(0).

If linear input throttling values are indicated by CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] then values from zero through the MBA_MAX field from CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] are supported as inputs. In the linear mode the input precision is defined as 100-(MBA_MAX). For instance, if the MBA_MAX value is 90, the input precision is 10%. Values not an even multiple of the precision (e.g., 12%) will be rounded down (e.g., to 10% delay applied).

- If linear values are not supported (CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] = 0) then input delay values are powers-of-two from zero to the MBA_MAX value from CPUID. In this case any values not a power of two will be rounded down the next nearest power of two.

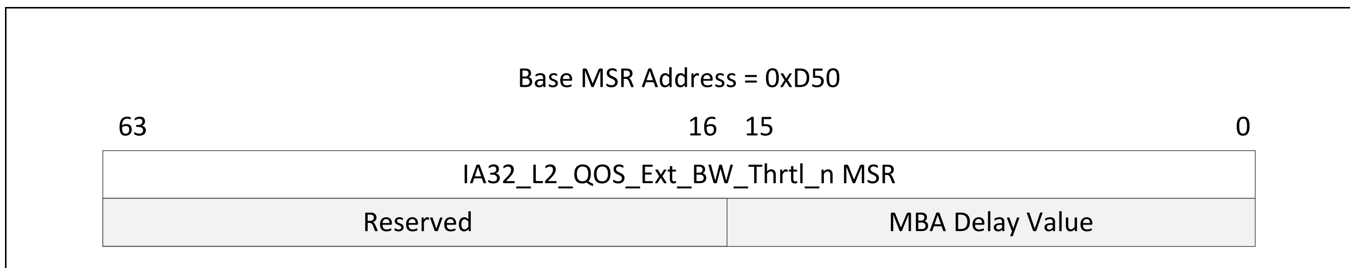


Figure 17-39. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition

Note that the throttling values provided to software are calibrated through specific traffic patterns, however as workload characteristics may vary the response precision and linearity of the delay values will vary across products, and should be treated as approximate values only.

17.18.7.3 Memory Bandwidth Allocation Usage Considerations

As the memory bandwidth control that MBA provides is indirect, using the feature with a closed-loop controller to also monitor memory bandwidth and how effectively the applications use the cache (via the Cache Monitoring Technology feature) may provide additional value. This approach also allows administrators to provide a bandwidth target or set-point which a controller could use to guide MBA throttling values applied, and this allows bandwidth control independent of the execution characteristics of the application.

As control is provided per processor core (the max of the delay values of the per-thread CLOS applied to the core) care should be taking in scheduling threads so as to not inadvertently place a high-priority thread (with zero intended MBA throttling) next to a low-priority thread (with MBA throttling intended), which would lead to inadvertent throttling of the high-priority thread.

12. Updates to Chapter 18, Volume 3B

Change bars show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Clock update to table 18-72 for Future Intel® Xeon® processors.

CHAPTER 18
PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

18.1 PERFORMANCE MONITORING OVERVIEW

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

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

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

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)." Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they are listed in Chapter 19, "Performance-Monitoring Events."

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 18.2.

See also:

— Section 18.2, "Architectural Performance Monitoring"

— Section 18.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)"

— Section 18.4, "Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)"

— Section 18.5, "Performance Monitoring (45 nm and 32 nm Intel® Atom™ Processors)"

— Section 18.6, "Performance Monitoring for Silvermont Microarchitecture"

— Section 18.7, "Performance Monitoring for Goldmont Microarchitecture"

— Section 18.8, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem"

— Section 18.8.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"

— Section 18.9, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge"

— Section 18.9.8, "Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility"

— Section 18.10, “3rd Generation Intel® Core™ Processor Performance Monitoring Facility”
— Section 18.11, “4th Generation Intel® Core™ Processor Performance Monitoring Facility”
— Section 18.12, “5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility”
— Section 18.13, “6th Generation Intel® Core™ Processor and 7th Generation Intel® Core™ Processor Performance Monitoring Facility”
— Section 18.14, “Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring”
— Section 18.15, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)”
— Section 18.16, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
— Section 18.20, “Performance Monitoring and Dual-Core Technology”
— Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache”
— Section 18.23, “Performance Monitoring (P6 Family Processor)”
— Section 18.24, “Performance Monitoring (Pentium Processors)”

18.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T 7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 1, 2, and 3. Intel processors based on the Skylake and Kaby Lake microarchitectures support versionID 4.

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support versionID 4.

18.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- Bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.

- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR)
- Number of bits supported in each IA32_PMCx
- Number of architectural performance monitoring events supported in a logical processor

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/ IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

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

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

See Figure 18-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7)** — Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

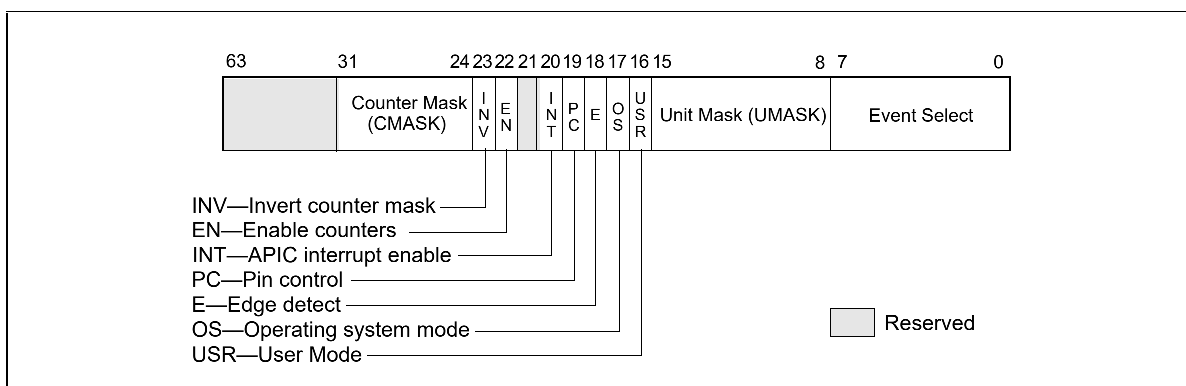


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX. Architectural performance events available in the initial implementation are listed in Table 19-1.
- **USR (user mode) flag (bit 16)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19)** — When set, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20)** — When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31)** — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

18.2.1.2 Pre-defined Architectural Performance Events

Table 18-1 lists architecturally defined events.

Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

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

Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

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

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

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

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

  — HLT

  — STPCLK# pin asserted

  — being throttled by TM1

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

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

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

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

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

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

  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

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

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

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

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

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

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

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

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

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

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

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

NOTE

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

18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.
- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

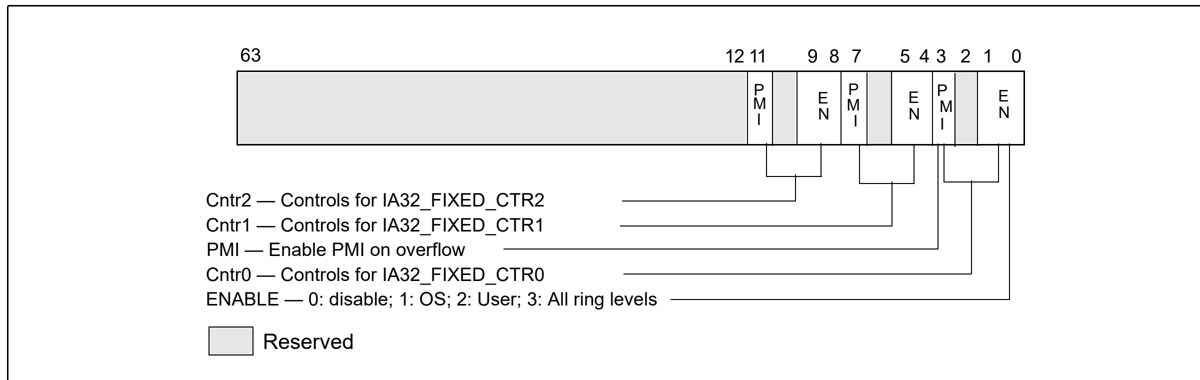


Figure 18-2. Layout of IA32_FIXED_CTR_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

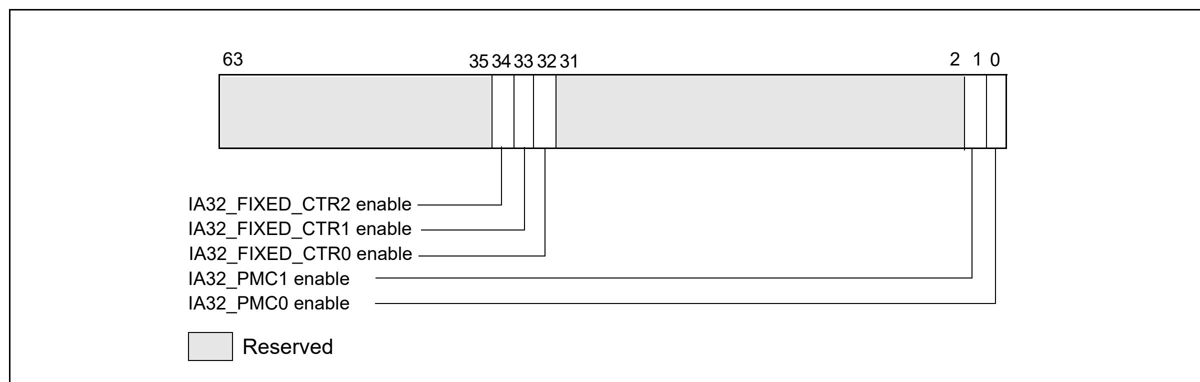


Figure 18-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

Table 18-2. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 18-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status., and sets the “OvfBuffer” bit in IA32_PERF_GLOBAL_STATUS.

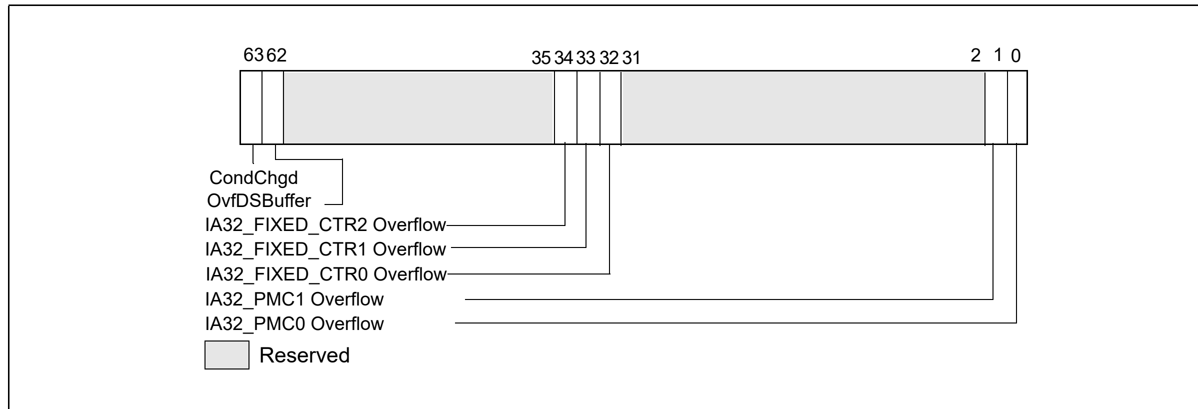


Figure 18-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 18-5.

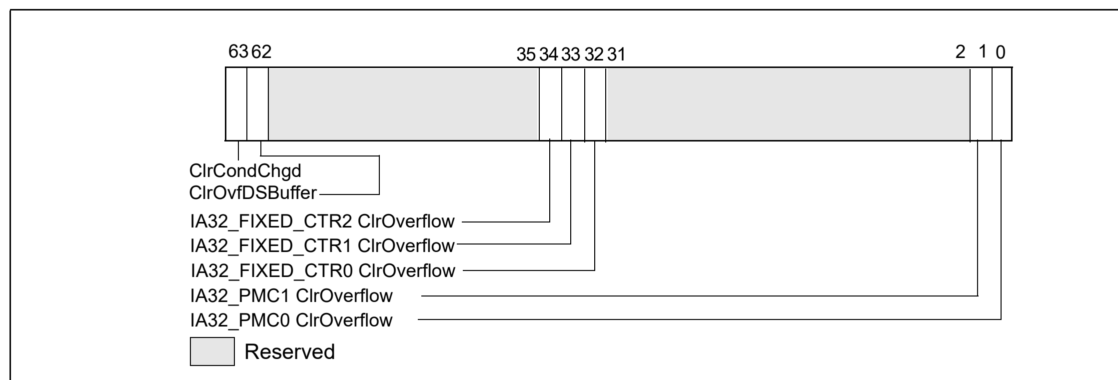


Figure 18-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

18.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- Anythread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  — Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 18-6.

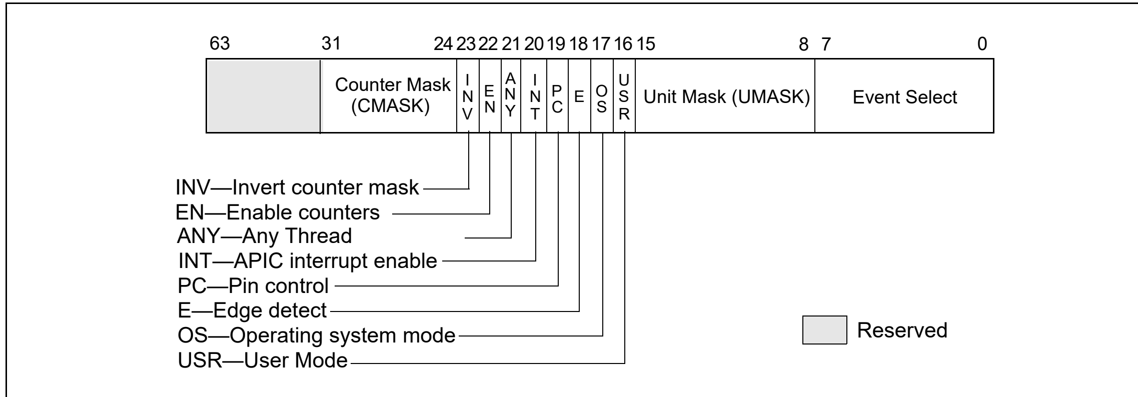


Figure 18-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

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

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

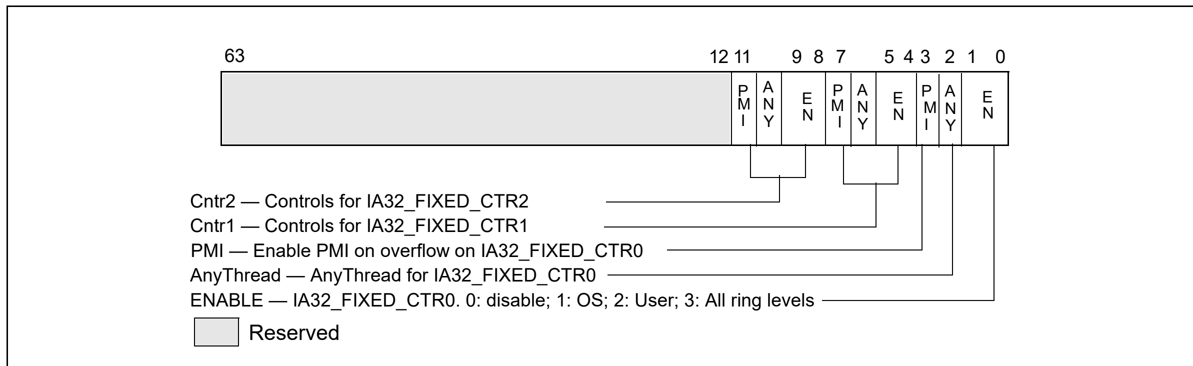


Figure 18-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

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

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 18-8 and Figure 18-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

  Note: The number of general-purpose performance monitoring counters (i.e. N in Figure 18-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g. N=2 for 45 nm Intel Atom processors; N =4 for processors based on the Nehalem microarchitecture; for processors based

on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active).

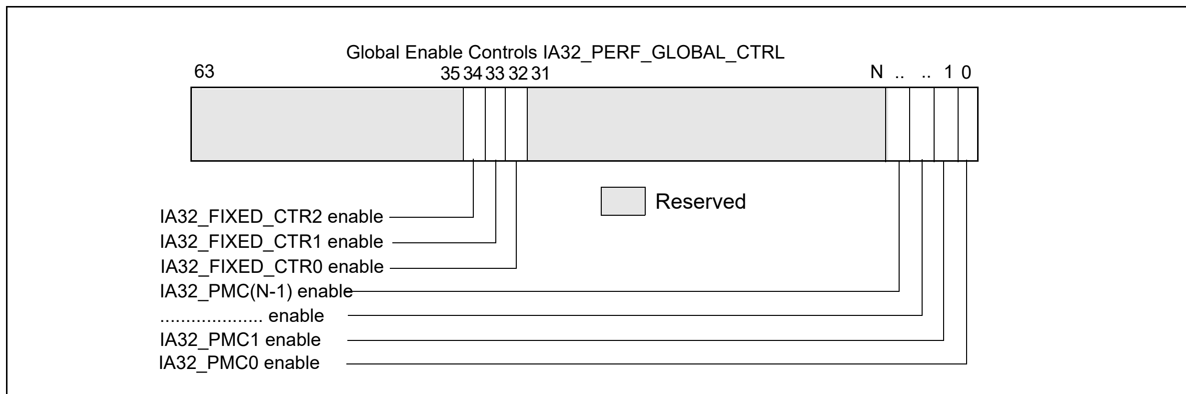


Figure 18-8. Layout of Global Performance Monitoring Control MSR

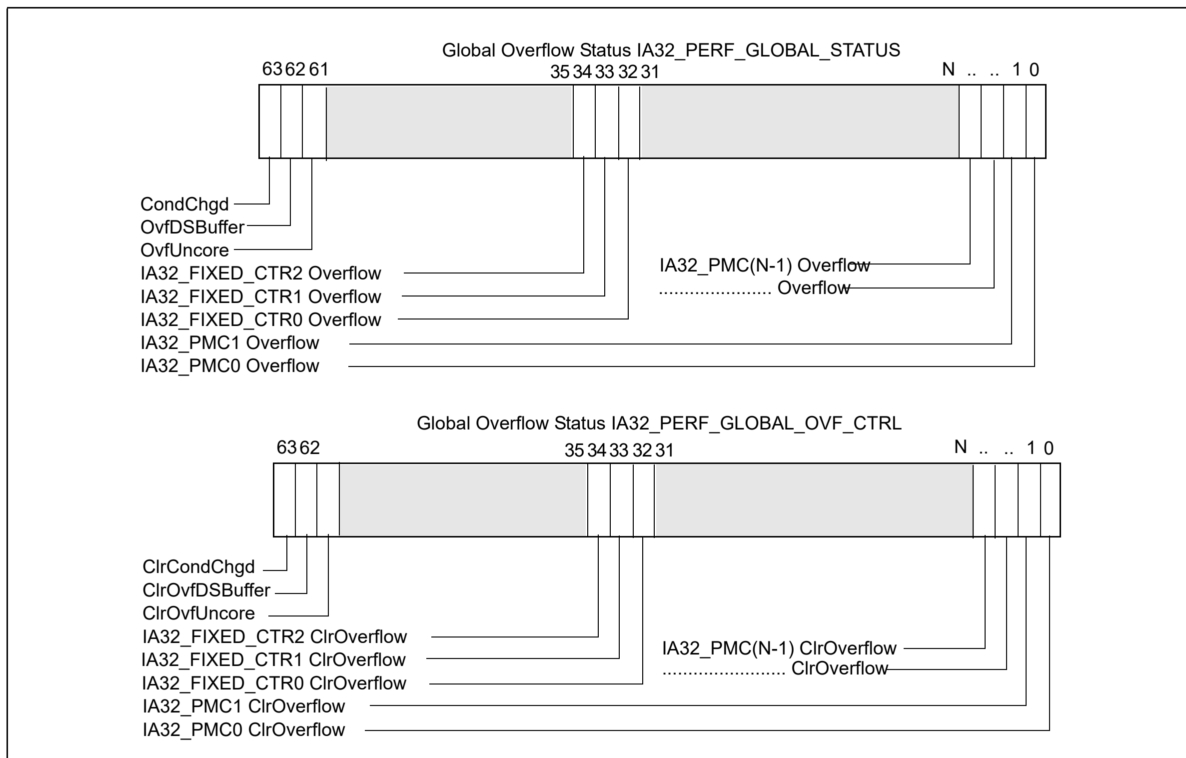


Figure 18-9. Global Performance Monitoring Overflow Status and Control MSRs

18.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While Anythread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 23, “Introduction to Virtual-Machine Extensions”) where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see CHAPTER 24 for additional information),
- clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see CHAPTER 24, CHAPTER 26, and CHAPTER 27).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted (see relevant sections of Chapter 19, “Performance Monitoring Events”).

18.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancement:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 18.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 18.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 18.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

18.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  — IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serve as an read-only control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:

  — (IA32_DEBUGCTL.LBR & (!IA32_PERF_GLOBAL_STATUS.LBR_FRZ)) =1
- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:
  — IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and
  — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- (IA32_PERFEVTSELn.EN & IA32_PERF_GLOBAL_CTRL.PMCn & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for programmable counter 'n', or
- (IA32_PERF_FIXED_CRTL.ENi & IA32_PERF_GLOBAL_CTRL.FCi & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for fixed counter 'i'

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeza_Perfmon_On_PMI to filter out data that may distort target workload analysis, see Table 17-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g. race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support SGX (For additional information about Intel SGX, see "Intel® Software Guard Extensions Programming Reference".):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e. IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

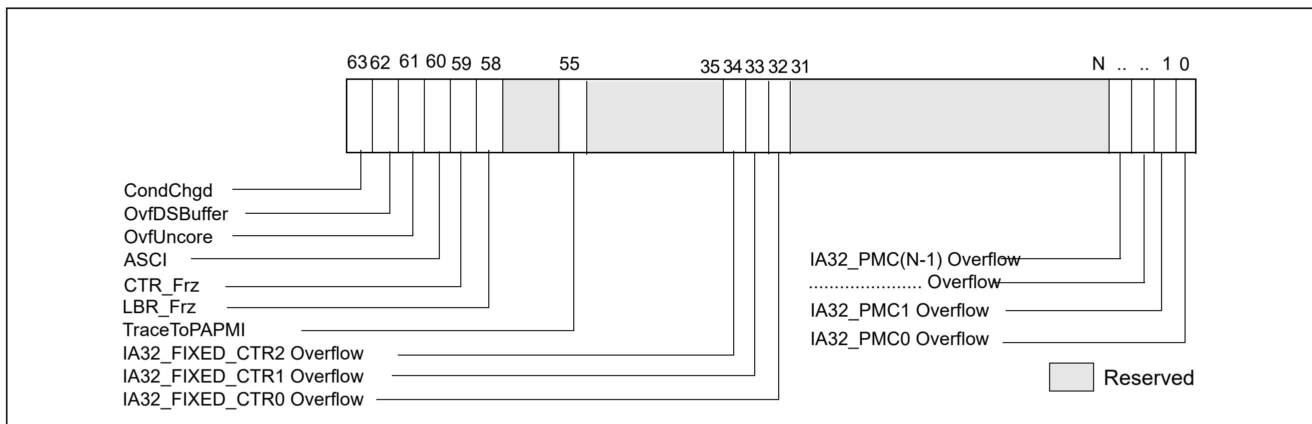


Figure 18-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural Perfmon Version 4

Note, a processor's support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

18.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_GLOBAL_STATUS_RESET (see Figure 18-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 18.2.4.1.

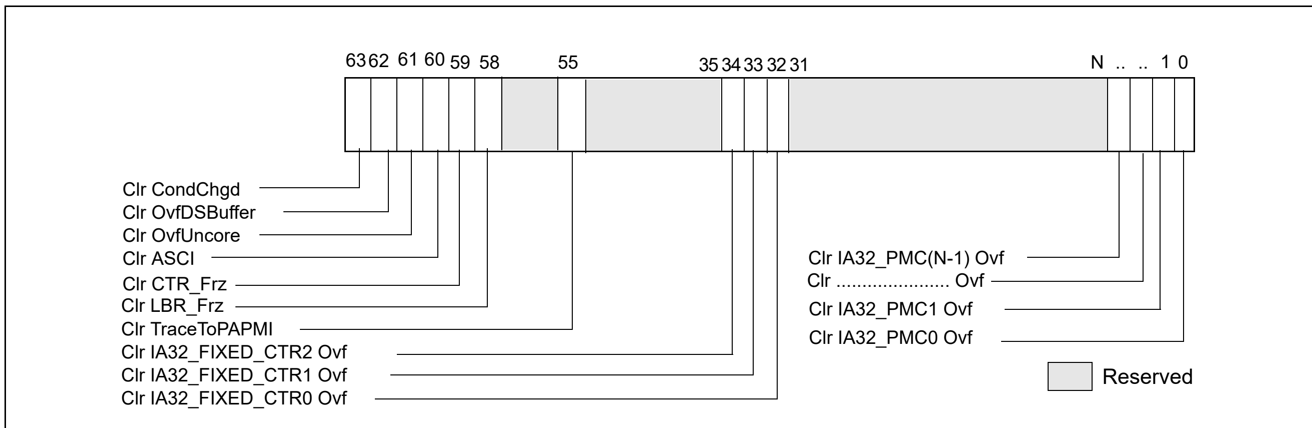


Figure 18-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

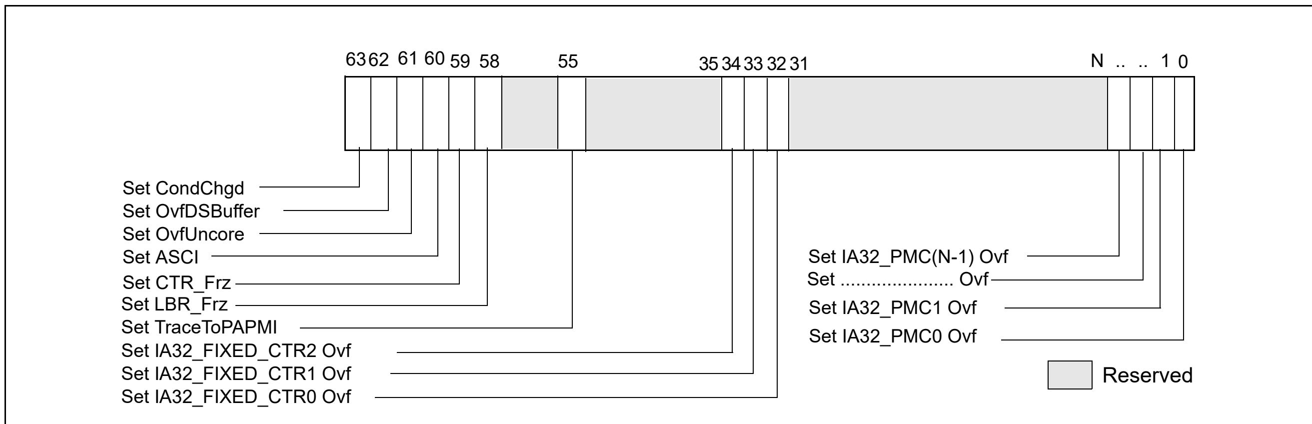


Figure 18-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4

18.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor's performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve the need of multiple agent adequately. A white paper, "Performance Monitoring Unit Sharing Guideline"[1], proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 18-13.

1. Available at http://www.intel.com/sdm

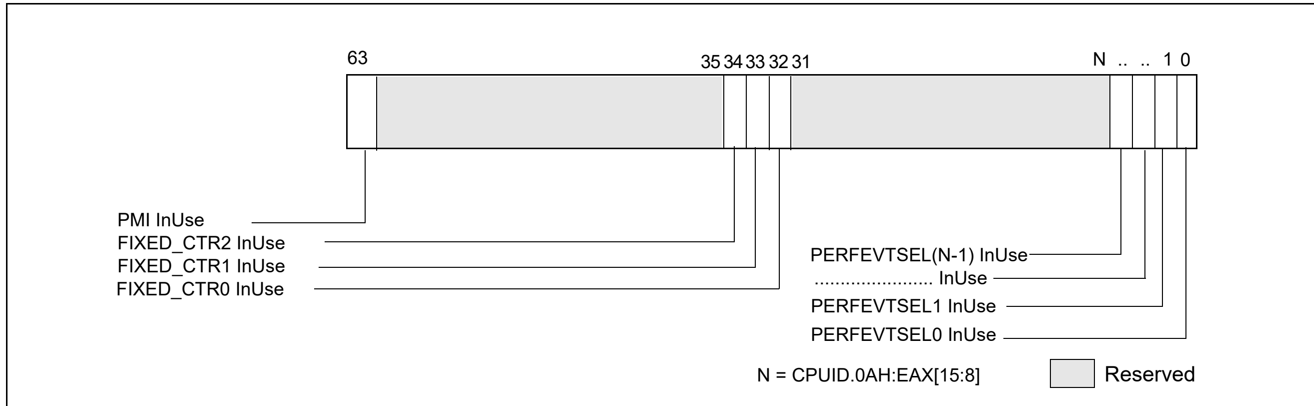


Figure 18-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_INUSE MSR provides an "InUse" bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL0_InUse[bit 0]: This bit reflects the logical state of (IA32_PERFEVTSEL0[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL1_InUse[bit 1]: This bit reflects the logical state of (IA32_PERFEVTSEL1[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL2_InUse[bit 2]: This bit reflects the logical state of (IA32_PERFEVTSEL2[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSELn_InUse[bit n]: This bit reflects the logical state of (IA32_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32_PERF_GLOBAL_INUSE.FC0_InUse[bit 32]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[1:0] != 0).
- IA32_PERF_GLOBAL_INUSE.FC1_InUse[bit 33]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[5:4] != 0).
- IA32_PERF_GLOBAL_INUSE.FC2_InUse[bit 34]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[9:8] != 0).
- IA32_PERF_GLOBAL_INUSE.PMI_InUse[bit 32]: This bit is set if any one of the following bit is set:
  — IA32_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8];
  — IA32_FIXED_CTR_CTRL.ENi_PMI, i = 0, 1, 2;
  — IA32_PEBS_ENABLES.EN_PMCi, i = 0, 1, 2, 3

18.2.5 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 18-49.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] ← EDX[COUNTERWIDTH-33:0]);
IA32_PMCi[31:0] ← EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved

18.3 PERFORMANCE MONITORING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 18-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 8.6, "Detecting Hardware Multi-Threading Support and Topology," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). As a result, the unit mask field (for example, IA32_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 18-3. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see Chapter 19, "Performance Monitoring Events") and for Intel Core Duo processors. Such events are referred to as core-specific events.

Table 18-3. Core Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	**Description**
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 18-4.

Table 18-4. Agent Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	**Description**
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 18-5.

Table 18-5. HW Prefetch Qualification Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	**Description**
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 18-6. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

Table 18-6. MESI Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

18.4 PERFORMANCE MONITORING (PROCESSORS BASED ON INTEL® CORE™ MICROARCHITECTURE)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 18.4.1). IA32_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 18-1. Starting with Intel Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 18.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields identical to those listed in Table 18-3, Table 18-4, Table 18-5, and Table 18-6. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-24 in Chapter 19, “Performance-Monitoring Events.”

In addition, the UMASK filed may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 18-7.

Table 18-7. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	HITM response
Bit 10	Reserved
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 18-8.

Table 18-8. Snoop Type Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	**Description**
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

18.4.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 18-2 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR MSR_PERF_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 18-14. Two sub-fields are defined for each control. See Figure 18-14; bit fields are:

- **Enable field (low 2 bits in each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

  When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

  Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

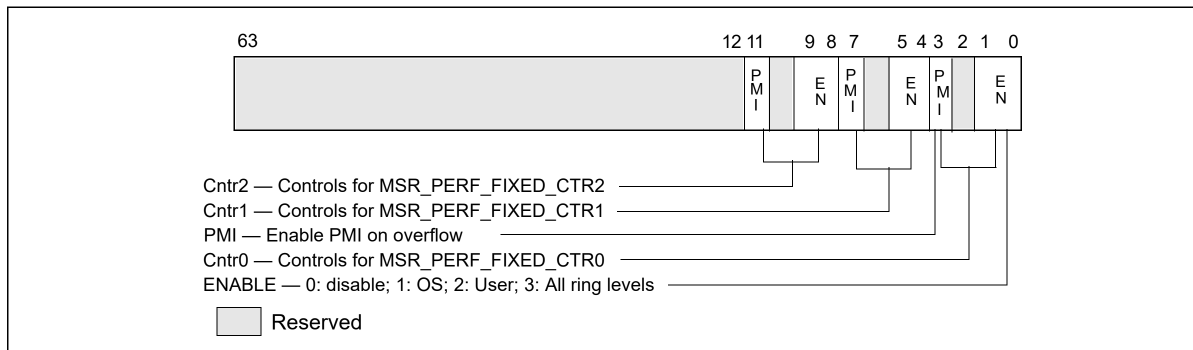


Figure 18-14. Layout of MSR_PERF_FIXED_CTR_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

18.4.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e. enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- MSR_PERF_GLOBAL_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 18-15). Each enable bit in MSR_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or MSR_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

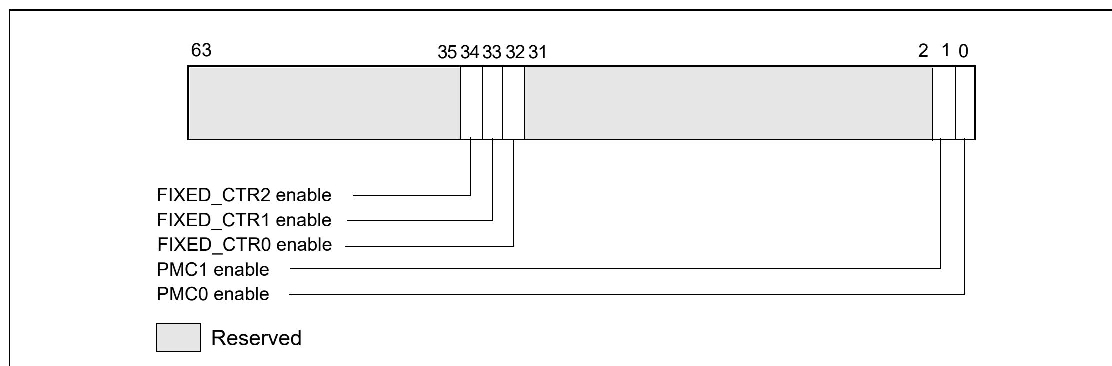


Figure 18-15. Layout of MSR_PERF_GLOBAL_CTRL MSR

MSR_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 18-16). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

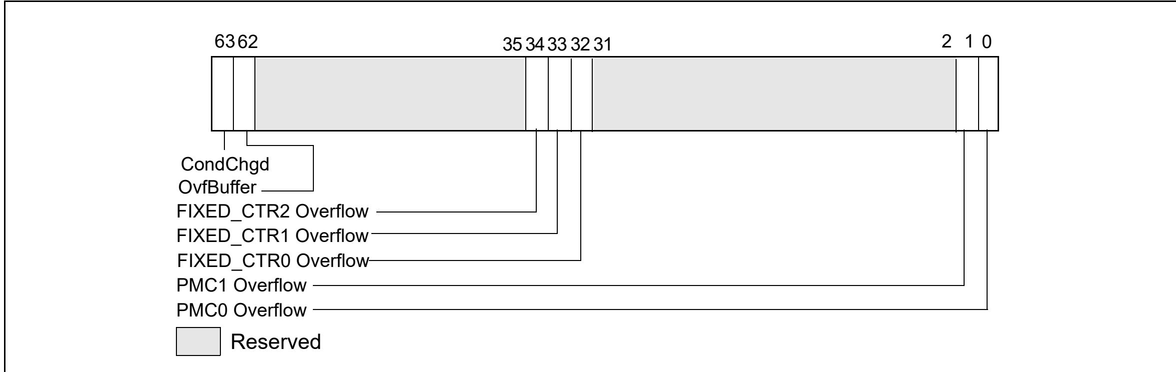


Figure 18-16. Layout of MSR_PERF_GLOBAL_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

MSR_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-17). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

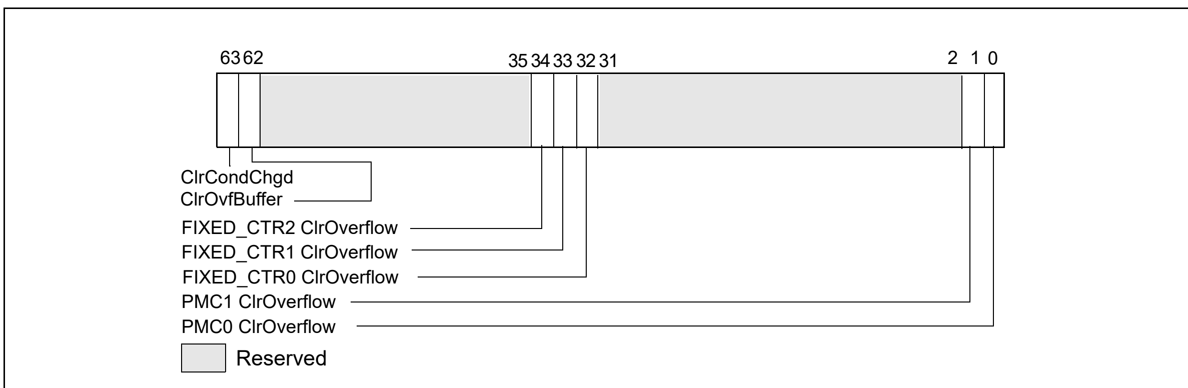


Figure 18-17. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

18.4.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst® microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 18.15.6, "At-Retirement Counting"), but is limited to IA32_PMC0. See Table 18-9 for a list of events available to processors based on Intel Core microarchitecture.

Table 18-9. At-Retirement Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

18.4.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4.2 and Section 17.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 18-10. The procedure for detecting availability of PEBS is the same as described in Section 18.15.7.1.

Table 18-10. PEBS Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

18.4.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 17-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 18-10.

18.4.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of IA32_PERF_CAPABILITES are defined in Table 2-2 of Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1
- PEBSRecordFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (seeSection 18.15.7).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (seeSection 18.8.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (seeSection 18.11.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32_PERF_GLOBAL_STATUS at offset 90H. (see Section 18.13.1.1).

18.4.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.1, "64 Bit Format of the DS Save Area," for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-11.

Table 18-11. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: ▪ Clear all counters if “Counter Freeze on PMI“ is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write “sample-after“ values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

NOTES:

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

18.4.4.4 Re-configuring PEBS Facilities

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

18.5 PERFORMANCE MONITORING (45 NM AND 32 NM INTEL® ATOM™ PROCESSORS)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 18.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-27.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32_PERFEVTSELx MSR, i.e. if IA32_PERFEVTSELx.AnyThread =1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 18-3, Table 18-4, Table 18-5, and Table 18-6. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-27 in Chapter 19, "Performance-Monitoring Events." Precise Event Based Monitoring is supported using IA32_PMC0 (see also Section 17.4.9, "BTS and DS Save Area").

18.6 PERFORMANCE MONITORING FOR SILVERMONT MICROARCHITECTURE

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-26.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

18.6.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

18.6.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32_PMC0 (see also Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 18-12.

Table 18-12. PEBS Performance Events for the Silvermont Microarchitecture

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-13, and each field in the PEBS record is 64 bits long.

Table 18-13. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

18.6.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-14 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

In the Silvermont microarchitecture, each MSR_OFFCORE_RSPx is shared by two processor cores.

Table 18-14. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 18-18 and Figure 18-19. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.6.3 for details.

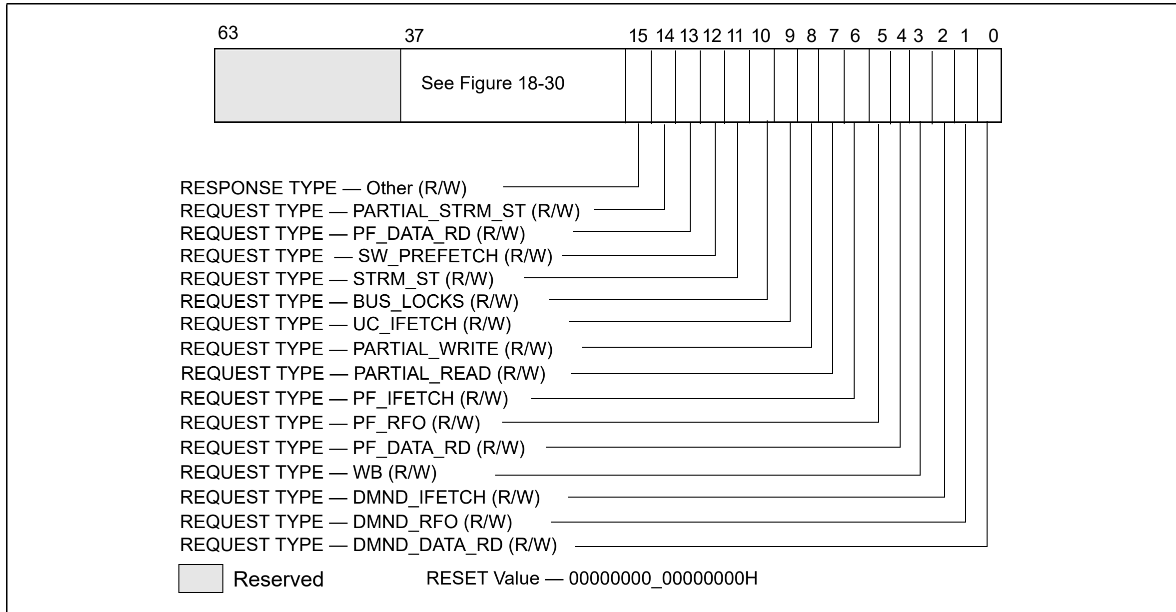


Figure 18-18. Request_Type Fields for MSR_OFFCORE_RSPx

Table 18-15. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	(R/W). Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	(R/W). Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	(R/W). Counts the number of UC instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
SW_PREFETCH	12	(R/W). Counts software prefetch requests
PF_DATA_RD	13	(R/W). Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	(R/W). Streaming store requests
ANY	15	(R/W). Any request that crosses IDI, including I/O.

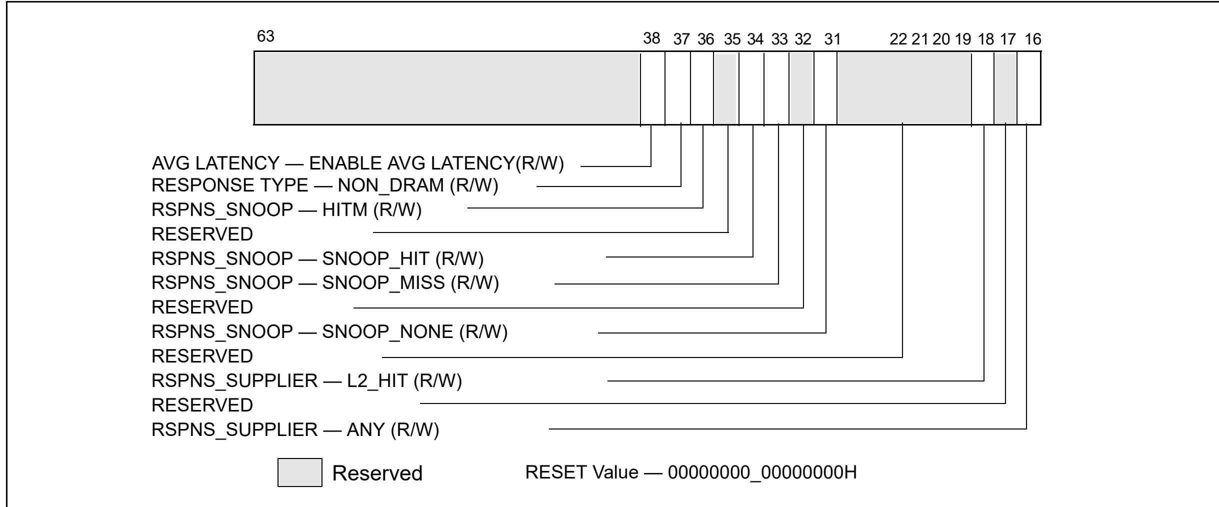


Figure 18-19. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 18-15) and a valid response type pattern (Table 18-16, Table 18-17). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-16. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	(R/W). Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	(R/W). Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.

Table 18-17. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	(R/W). Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	(R/W). Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved
	HITM	36	(R/W). Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	(R/W). Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0). This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

18.6.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR_OFFCORE_RSP registers. Using two performance monitoring counters, program the two OFFCORE_RESPONSE event encodings into the corresponding IA32_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, white setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

18.7 PERFORMANCE MONITORING FOR GOLDMONT MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont microarchitecture. They report architectural performance monitoring versionID = 4 (see Section 18.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Goldmont microarchitecture ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR.

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 18.6 , with some differences and enhancements summarized in Table 18-18.

Table 18-18. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	4	2	
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 18.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz	▪ Individual counter overflow ▪ PEBS buffer overflow	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	No	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 18.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-19.	See Section 18.6.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-12).	IA32_PMC0 only.
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 18-20; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	Table 18-13.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

18.7.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32_PMC0 for all events (see Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way as on the Silvermont microarchitecture. They can capture precise eventingIP associated with a retired instruction that caused the event. The PEBS record provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4 and Section 17.4.9). The PEBS record also provides architectural state of the processor after the instruction that caused the event completes. The list of precise events supported in the Goldmont microarchitecture is shown in Table 18-19.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. Other examples of a non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. There may be many instructions in various stages of execution, multiple or zero instructions being retired each cycle as CPU_CLK_UNHALTED.CORE_P increments. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. The PEBS record will be generated at the next opportunity, capturing the processor state when the machine received the interrupt, even if interrupts are masked. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32_PERF_GLOBAL_STATUS_SET.

Table 18-19. Precise Events Supported by the Goldmont Microarchitecture

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	C0H	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH

Table 18-19. Precise Events Supported by the Goldmont Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-13, and each field in the PEBS record is 64 bits long.

Table 18-20. PEBS Record Format for the Goldmont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	C0H	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the "applicable counter" field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

18.7.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling. The fields are present in the record but are reserved.

For Goldmont, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g. the one that triggered a precise event that caused the PEBS record to be generated).

18.7.1.2 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the "skid" problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 18.9.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g. via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g. what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

18.7.1.3 Enhancements to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62]

In addition to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62] being set when PEBS_Index reaches the PEBS_Interrupt_Theshold, the bit is also set when PEBS_Index is out of bounds. That is, the bit will be set when PEBS_Index < PEBS_Buffer_Base or PEBS_Index > PEBS_Absolute_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32_PERF_GLOBAL_STATUS will be cleared according to Applicable Counters, however the IA32_PMCx values will not be reloaded with the Reset values stored in the DS_AREA.

18.7.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-14 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR_OFFCORE_RSPx registers per core.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 18-21.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-16.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-22.

- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 18.6.3 for details.

Table 18-21. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	(R/W) Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	(R/W) Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	(R/W) Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	(R/W) Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	(R/W) Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	(R/W) Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	(R/W) Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	(R/W) Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	(R/W) Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	(R/W) Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	(R/W) Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	(R/W) Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	(R/W) Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	(R/W) Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	(R/W) Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-15) and a valid response type pattern (either Table 18-16 or Table 18-22). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-22. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	(R/W). A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	(R/W) A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	(R/W) Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	(R/W) Target was a non-DRAM system address. This includes MMIO transactions.

Table 18-22. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests (Contd.)

Subtype	Bit Name	Offset	Description
Outstanding requests[1]	OUTSTANDING	38	(R/W) Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

NOTES:

1. See Section 18.6.3, "Average Offcore Request Latency Measurement" for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

[ANY 'OR' (L2 Hit)] 'XOR' (Snoop Info Bits) 'XOR' (Avg Latency)

18.7.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 18.6.3.

18.8 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Intel Core i7 processor family[2] supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-27. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

2. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem, so the performance monitoring facilities described in this section generally also apply.

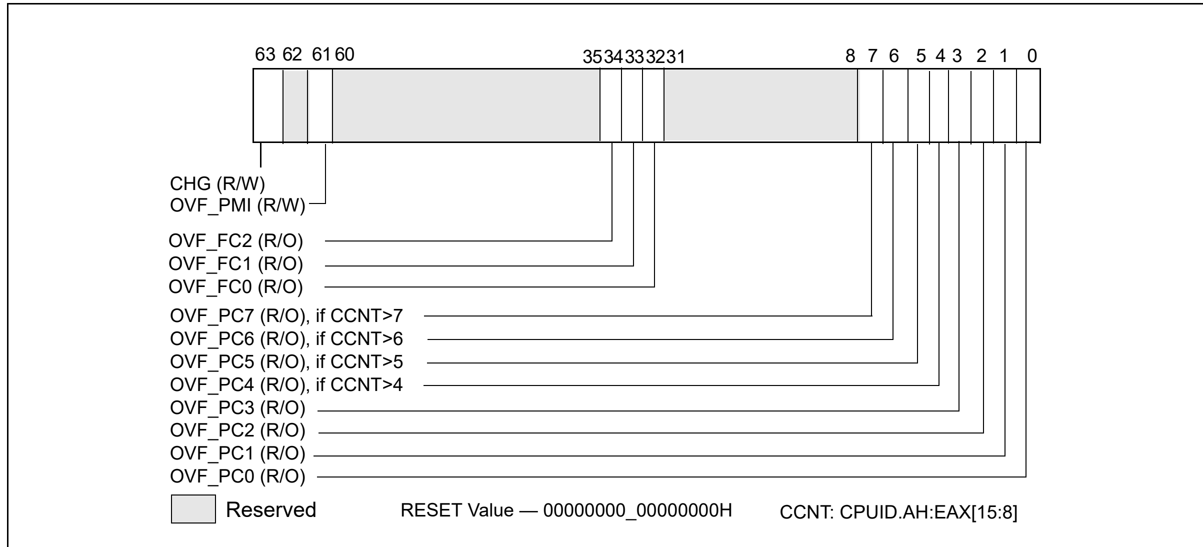


Figure 18-20. IA32_PERF_GLOBAL_STATUS MSR

18.8.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Intel micro-architecture code name Nehalem has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Intel microarchitecture code name Nehalem. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

18.8.1.1 Processor Event Based Sampling (PEBS)

All four general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 18-21.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

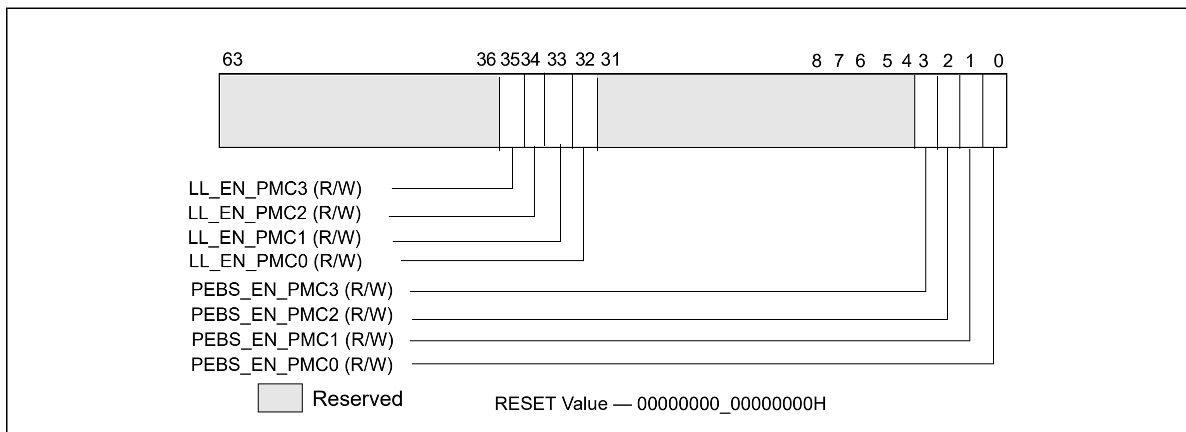


Figure 18-21. Layout of IA32_PEBS_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8] (see Figure 18-49).

The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 18-49). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

mov eax, [eax] ; causes PEBS assist

nop

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 18-23, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-23. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 18-10. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PEBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS_BUFFER_MANAGEMENT_AREA data structure must be programmed into the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 18-22.

- **PEBS Buffer Base**: This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer. Software should allocate this memory from the non-paged pool.
- **PEBS Index**: This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum**: This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

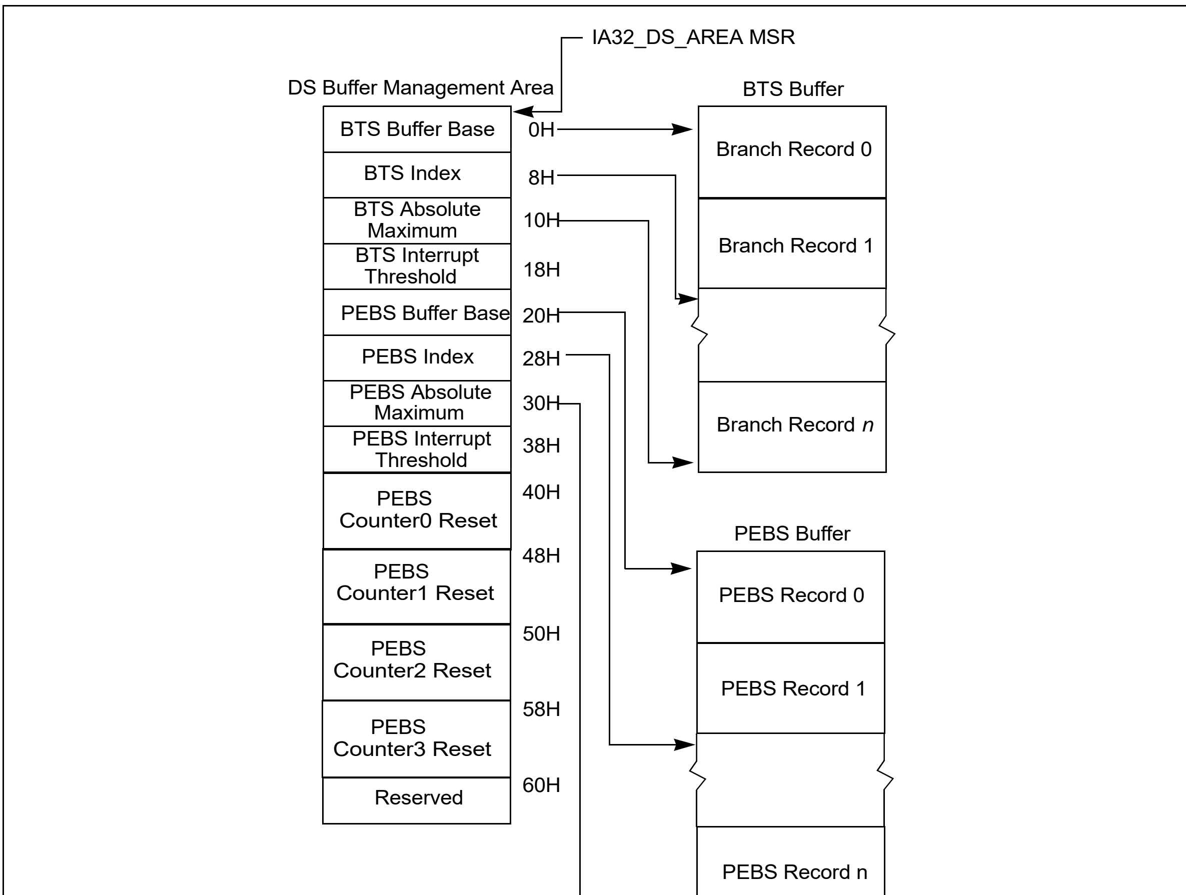


Figure 18-22. PEBS Programming Environment

- **PEBS Interrupt Threshold**: This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32_PERF_GLOBAL_STATUS.PEBS_Ovf bit will be set.
- **PEBS CounterX Reset**: This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

Performance Counter Prioritization

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32_PerfEvtSelX.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32_PMC0 takes precedence over all other counters. Counter IA32_PMC1 takes precedence over counters IA32_PMC2 and IA32_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32_PMC1 cause an overflow interrupt and IA32_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 18.8.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

18.8.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-23. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 18-23, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- Data Source: The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 18-24. In the descriptions local memory refers to system memory physically attached to a processor package, and remote memory referrals to system memory physically attached to another processor package.

Table 18-24. Data Source Encoding for Load Latency Record

Encoding	Description
00H	Unknown L3 cache miss
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
07H[1]	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies found
08H	L3 MISS. Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation
0FH	The request was to un-cacheable memory.

NOTES:

1. Bit 7 is supported only for processor with CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 18-23.

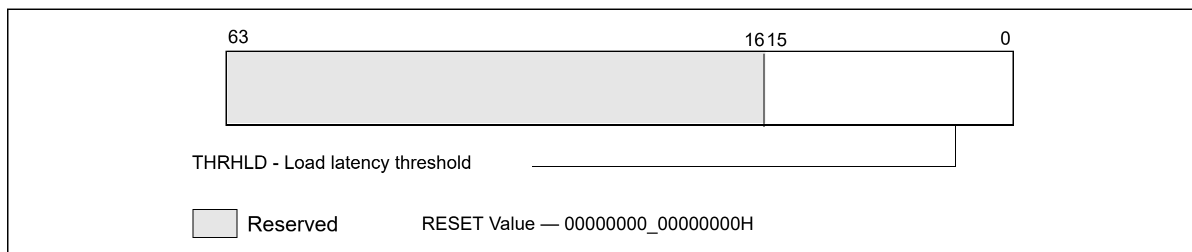


Figure 18-23. Layout of MSR_PEBS_LD_LAT MSR

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

18.8.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_0. There is only one off-core response configuration MSR. Table 18-25 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-25. Off-Core Response Event Encoding

Event code in IA32_PERFEVTSELx	Mask Value in IA32_PERFEVTSELx	Required Off-core Response MSR
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR_OFFCORE_RSP_0 is shown in Figure 18-24. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.

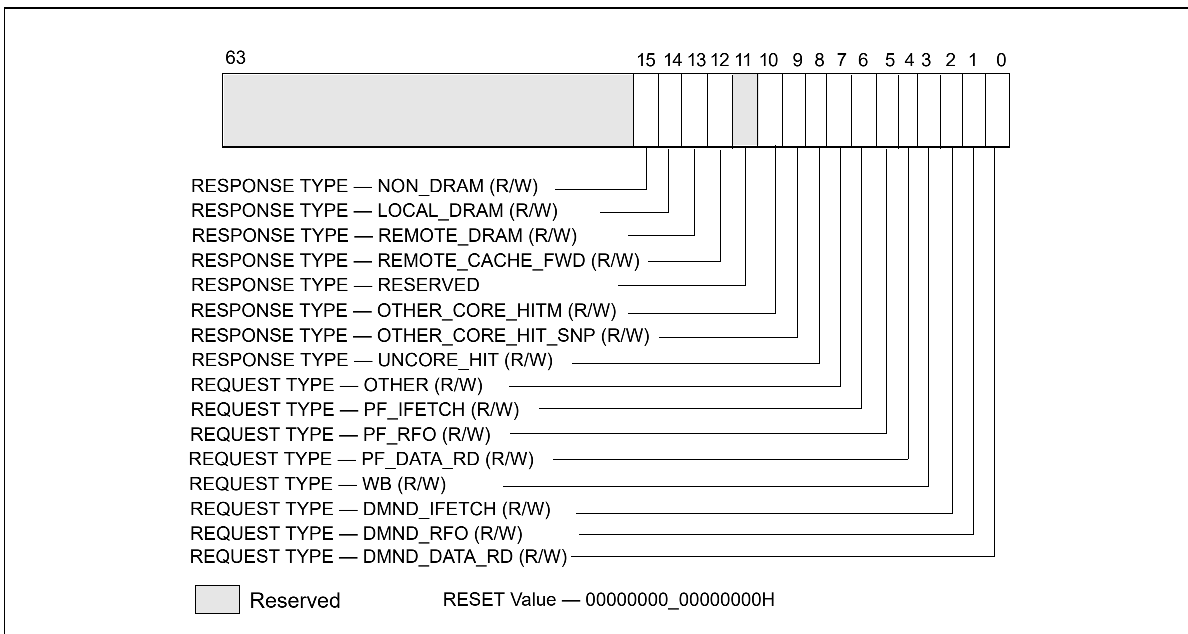


Figure 18-24. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 18-26. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.

Table 18-26. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
OTHER	7	(R/W). Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HI T_SNP	9	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HI TM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_ FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

18.8.2 Performance Monitoring Facility in the Uncore

The "uncore" in Intel microarchitecture code name Nehalem refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06_1AH, 06_1EH, 06_1FH (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR_UNCORE_PerfCntr0 through MSR_UNCORE_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR_UNCORE_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR_UNCORE_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

  The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

18.8.2.1 Uncore Performance Monitoring Management Facility

MSR_UNCORE_PERF_GLOBAL_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 18-25 shows the layout of MSR_UNCORE_PERF_GLOBAL_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR_UNCORE_PerfCntr n.
- EN_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR_UNCORE_FixedCntr0.

- EN_PMI_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32_DEBUGCTL.Offcore_PMI_EN to 1.
- PMI_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR_UNCORE_PERF_GLOBAL_CTRL upon exit from the ISR.

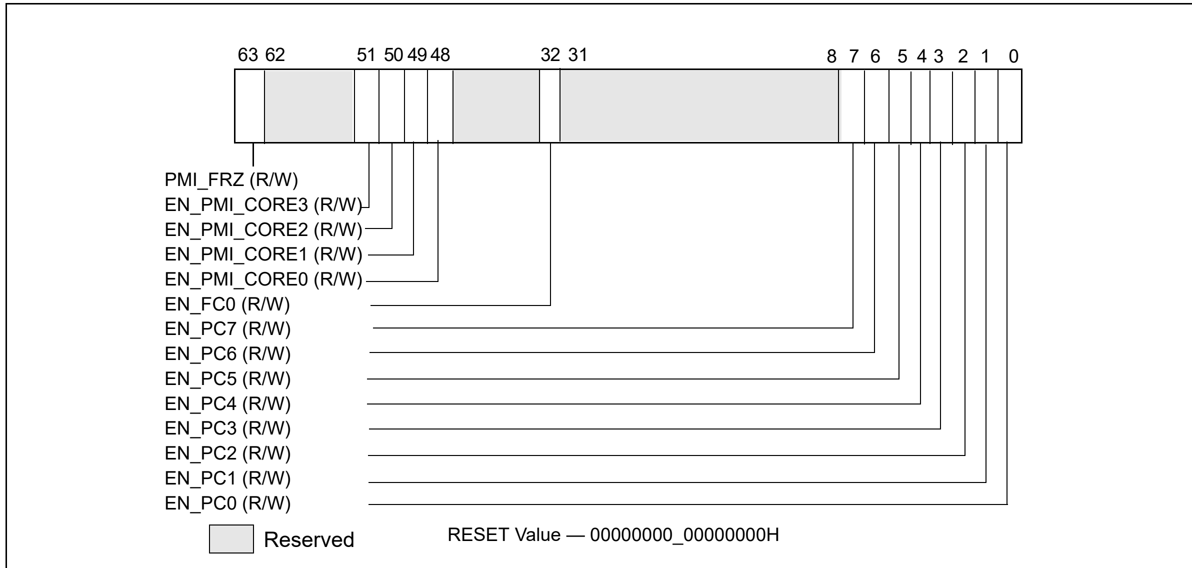


Figure 18-25. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

MSR_UNCORE_PERF_GLOBAL_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 18-26 shows the layout of MSR_UNCORE_PERF_GLOBAL_STATUS.

- OVF_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR_UNCORE_PerfCntr n has overflowed.
- OVF_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR_UNCORE_FixedCntr0 has overflowed.
- OVF_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR_UNCORE_PERF_GLOBAL_STATUS register has changed state.

MSR_UNCORE_PERF_GLOBAL_OVF_CTRL allows software to clear the status bits in the UNCORE_PERF_GLOBAL_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

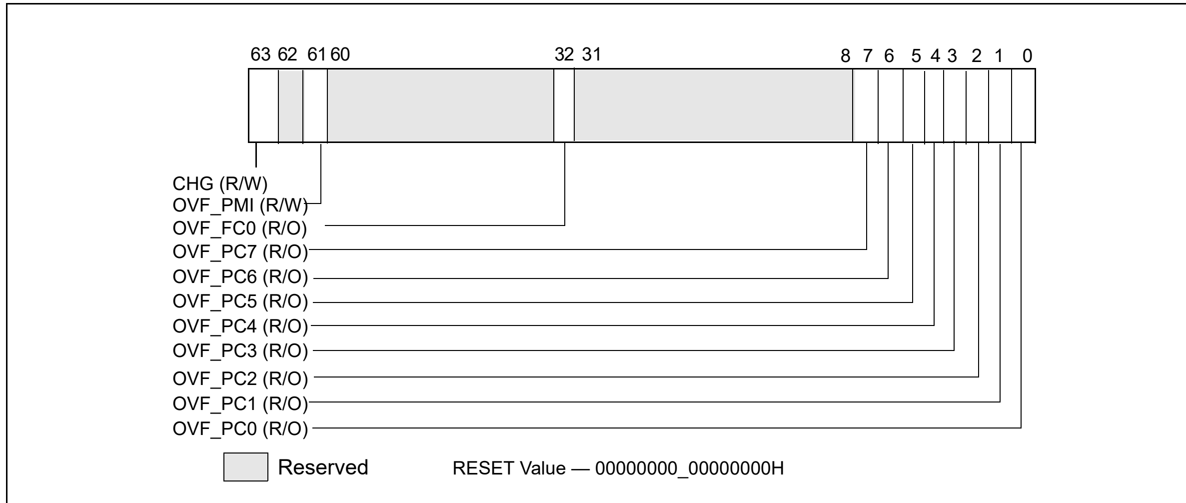


Figure 18-26. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 18-27 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

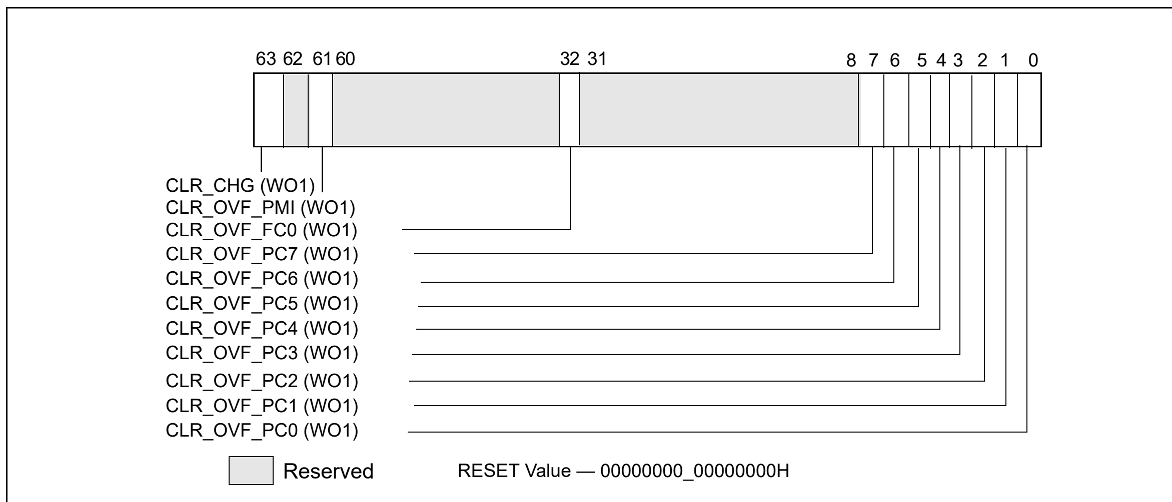


Figure 18-27. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- CLR_OVF_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR_UNCORE_PerfCntr n. Writing a value other than 1 is ignored.
- CLR_OVF_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR_UNCORE_FixedCntr0. Writing a value other than 1 is ignored.
- CLR_OVF_PMI (bit 61): Set this bit to clear the OVF_PMI flag in MSR_UNCORE_PERF_GLOBAL_STATUS. Writing a value other than 1 is ignored.
- CLR_CHG (bit 63): Set this bit to clear the CHG flag in MSR_UNCORE_PERF_GLOBAL_STATUS register. Writing a value other than 1 is ignored.

18.8.2.2 Uncore Performance Event Configuration Facility

MSR_UNCORE_PerfEvtSel0 through MSR_UNCORE_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR_UNCORE_PerfEvtSelx and counting must be enabled using the respective EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL. Figure 18-28 shows the layout of MSR_UNCORE_PERFEVTSELx.

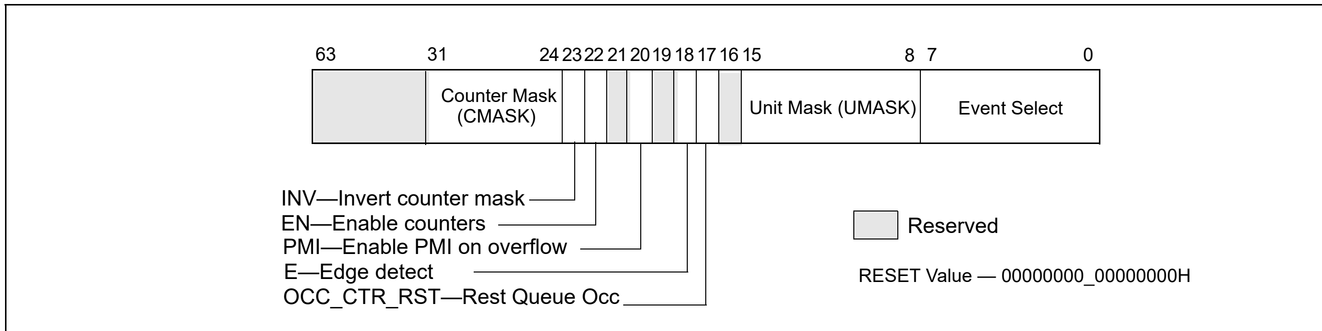


Figure 18-28. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC_CTR_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 18-29 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

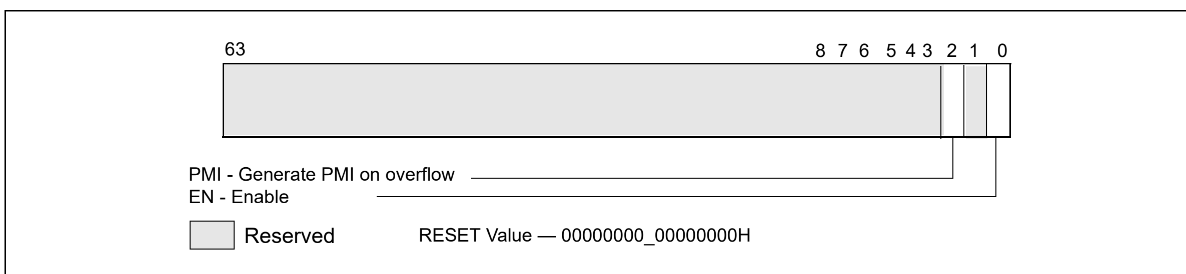


Figure 18-29. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN_FC0 bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.

Both the general-purpose counters (MSR_UNCORE_PerfCntr) and the fixed-function counter (MSR_UNCORE_FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

18.8.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR_UNCORE_PERFEVTSELx is used to select different uncore event logic unit. When the event “ADDR_OPCODE_MATCH” is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR_UNCORE_ADDR_OPCODE_MATCH register. The layout is shown in Figure 18-30.

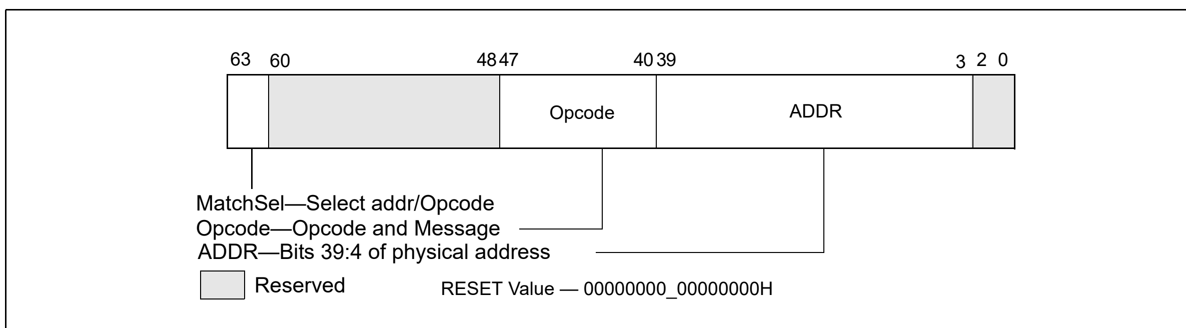


Figure 18-30. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- Addr (bits 39:3): The physical address to match if “MatchSel“ field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - Bits 43:40 specify the QPI packet header opcode,
  - Bits 47:44 specify the QPI message classes.

  Table 18-27 lists the encodings supported in the opcode field.

Table 18-27. Opcode Field Encoding for MSR_UNCORE_ADDR_OPCODE_MATCH

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr_opcode match hardware
  - 100B: Count if only the address field matches,
  - 010B: Count if only the opcode field matches
  - 110B: Count if either opcode field matches or the address field matches
  - 001B: Count only if both opcode and address field match
  - Other encoding are reserved

18.8.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 18-31.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

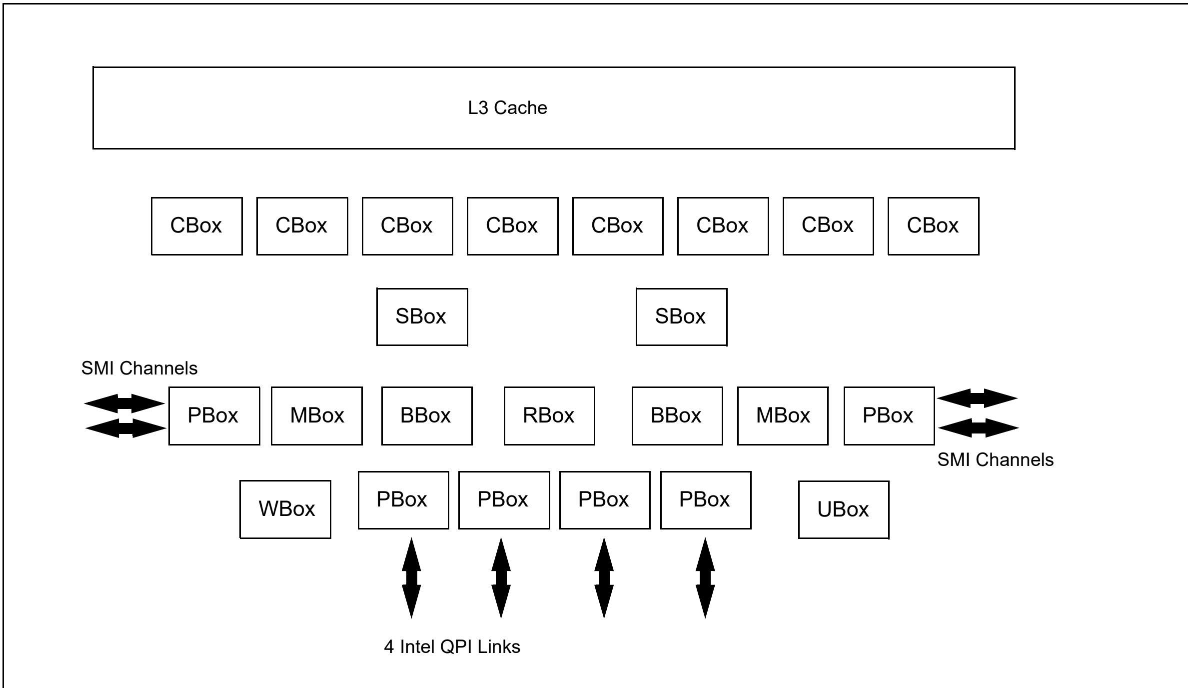


Figure 18-31. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-28 summarizes the number MSRs for uncore PMU for each box.

Table 18-28. Uncore PMU MSR Summary

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

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the "global control" programming interface, IA32_PERF_GLOBAL_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR_U_PMON_GLOBAL_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, Table 2-15 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more the one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_PMON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_CTR, MSR_S0_PMON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_EVNT_SEL, MSR_S0_PMON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR_M0_PMON_TIMESTAMP, MSR_R0_PMON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document “Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide“.

18.8.4 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 18.8 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 18-25 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Intel microarchitecture code name Westmere. The event code and event mask definitions of Non-architectural performance monitoring events are listed in Table 19-27.

The load latency facility is the same as described in Section 18.8.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 18-33.

18.8.5 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series[3]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore sub-system is similar to that shown in Figure 18-31, with the additional capability that up to 10 C-Box units are supported.

Table 18-29 summarizes the number MSRs for uncore PMU for each box.

Table 18-29. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None

3. Exceptions are indicated for event code 0FH in Table 19-20; and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 18-33.

Table 18-29. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

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

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the "Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual".

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

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Intel microarchitecture code name Sandy Bridge; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU's capability is similar to those described in Section 18.8.1 and Section 18.8.4, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 18-30.

Table 18-30. Core PMU Comparison

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-32.	See Table 18-10.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.9.4.2; ▪ Data source encoding ▪ STLB miss encoding ▪ Lock transaction encoding	Data source encoding	
PEBS-Precise Store	Section 18.9.4.3	No	

Table 18-30. Core PMU Comparison (Contd.)

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

18.9.1 Global Counter Control Facilities In Intel® Microarchitecture Code Name Sandy Bridge

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Intel microarchitecture code name Sandy Bridge. Software must use CPUID to determine the number performance counters/event select registers (See Section 18.2.1.1).

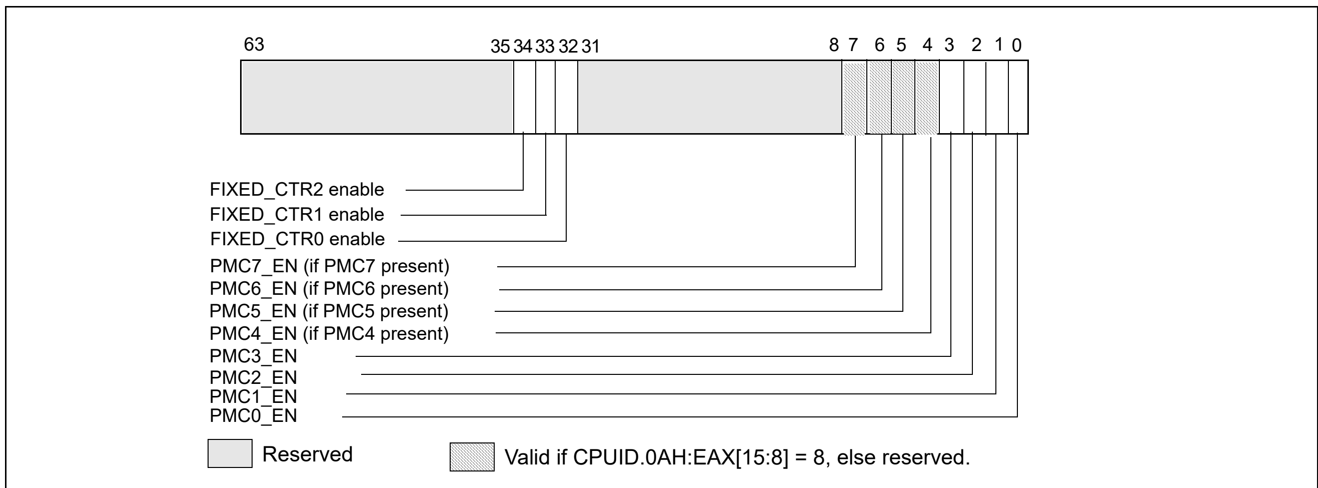


Figure 18-32. IA32_PERF_GLOBAL_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

Figure 18-15 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.
IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 18-33). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

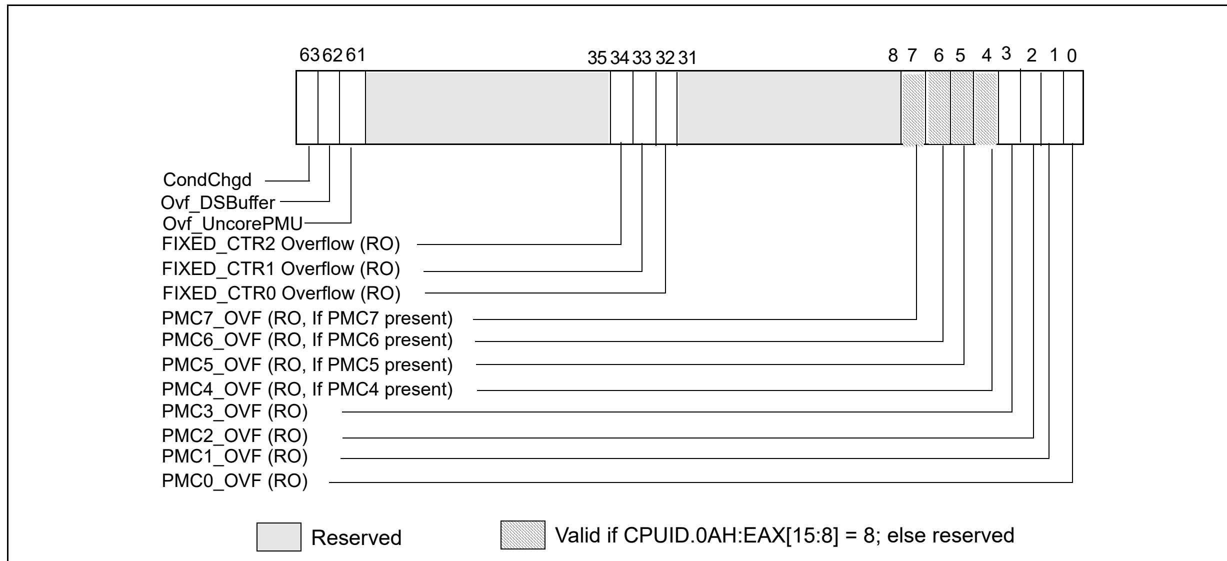


Figure 18-33. IA32_PERF_GLOBAL_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-34). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling
- Reloading counter values to continue sampling
- Disabling event counting or interrupt based sampling

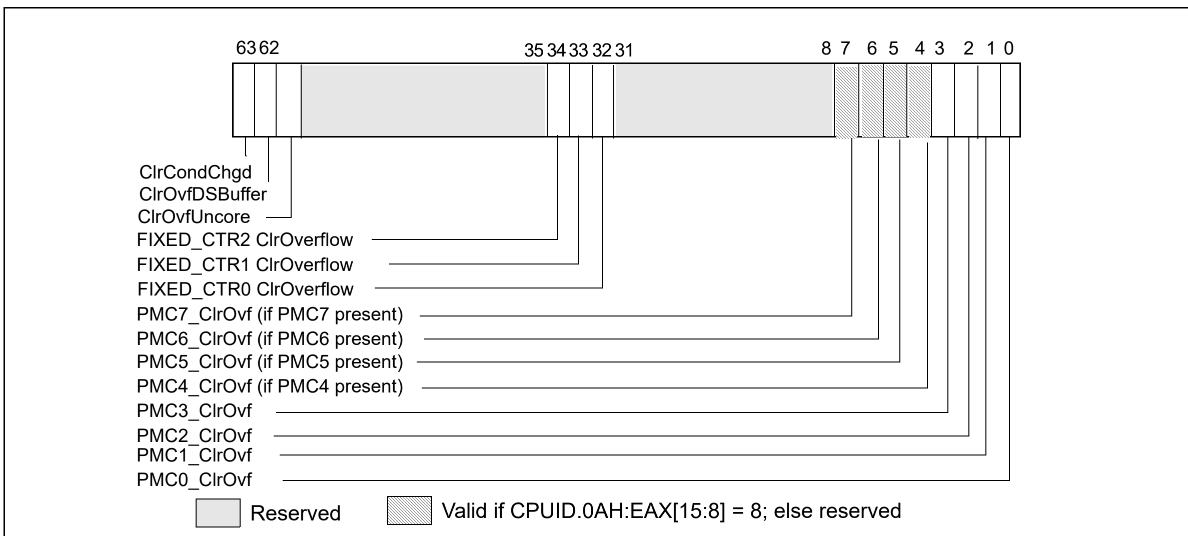


Figure 18-34. IA32_PERF_GLOBAL_OVF_CTRL MSR in Intel microarchitecture code name Sandy Bridge

18.9.2 Counter Coalescence

In processors based on Intel microarchitecture code name Sandy Bridge, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report either 4 or 8 depending specific processor's product features.

If a processor core is shared by two logical processors, each logical processors can access 4 counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, all eight general-purpose counters are visible, and CPUID.0AH:EAX[15:8] reports 8. IA32_PMC4-IA32_PMC7 occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_GLOBAL_OVF_CTL will also cause #GP.

18.9.3 Full Width Writes to Performance Counters

Processors based on Intel microarchitecture code name Sandy Bridge support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 18.2.4).

The default behavior of IA32_PMCx is unchanged, i.e. WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

18.9.4 PEBS Support in Intel® Microarchitecture Code Name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Westmere is summarized in Table 18-31.

Table 18-31. PEBS Facility Comparison

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-35	Figure 18-21	
PEBS record layout	Physical Layout same as Table 18-23.	Table 18-23	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 18-32.	See Table 18-10.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-33.	Table 18-24	
PEBS-Precise Store	Yes; see Section 18.9.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32_PEBS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32_PEBS_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.

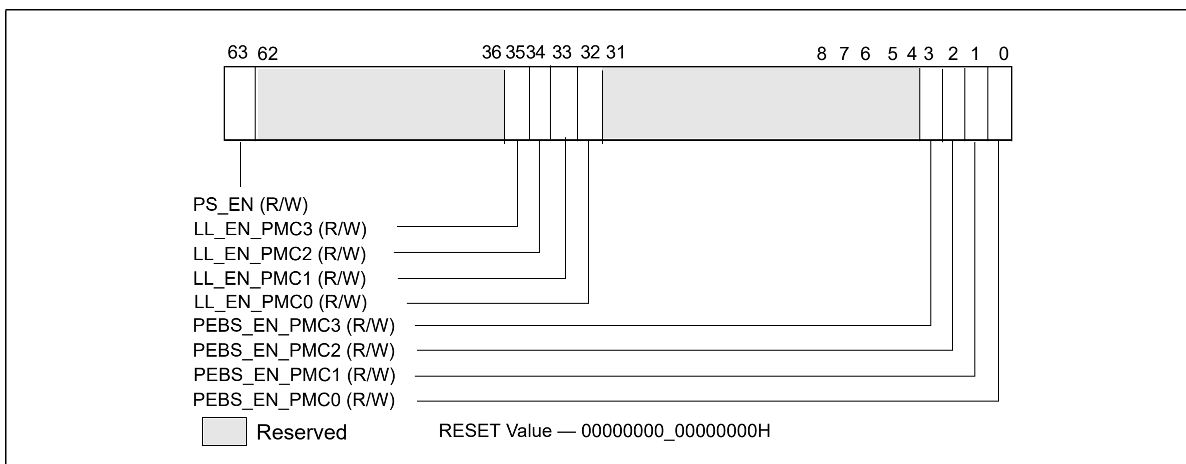


Figure 18-35. Layout of IA32_PEBS_ENABLE MSR

18.9.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 18-23, but the fields at offset 98H, A0H and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H):When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 18-24. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Intel microarchitecture code name Sandy Bridge is shown in Table 18-32.

Table 18-32. PEBS Performance Events for Intel® Microarchitecture Code Name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

NOTES:

1. Only available on IA32_PMC1.

18.9.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Intel microarchitecture code name Sandy Bridge is similar to that in prior microarchitecture. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-23 and Section 18.9.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the

programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-23. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 18-33. Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 18-24
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 18-23.

18.9.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-23. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 18-34. Layout of Precise Store Information In PEBS Record

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	**L1D Hit** (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. **STLB Miss** (bit 4): The store missed the STLB if set, otherwise the store hit the STLB **Locked Access** (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

18.9.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable “skids” in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the “skid” problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus eliminating skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

18.9.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 18-35 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-35. Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 18-36 and Figure 18-37. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

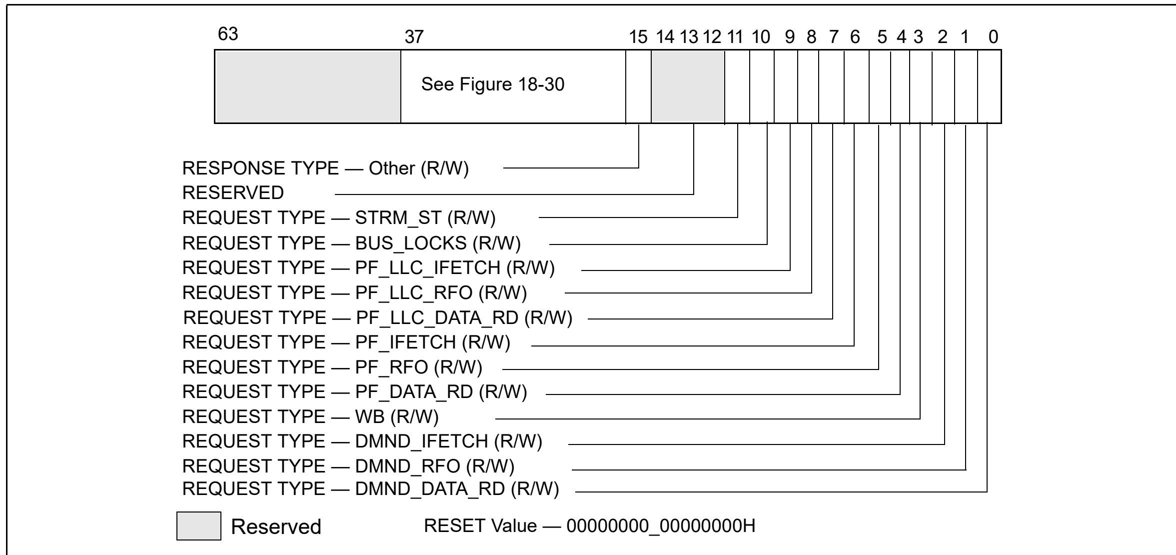


Figure 18-36. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 18-36. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	(R/W). L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	(R/W). RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	(R/W). L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

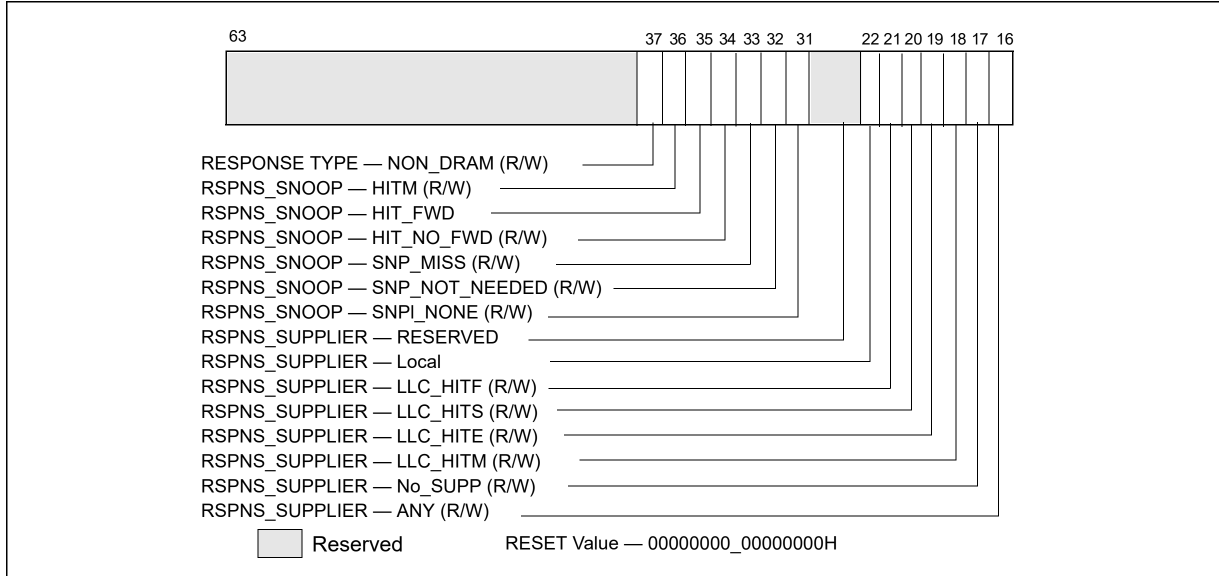


Figure 18-37. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSP_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-37. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

Table 18-38. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	SNP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

18.9.6 Uncore Performance Monitoring Facilities In Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 18.8.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 18-38.

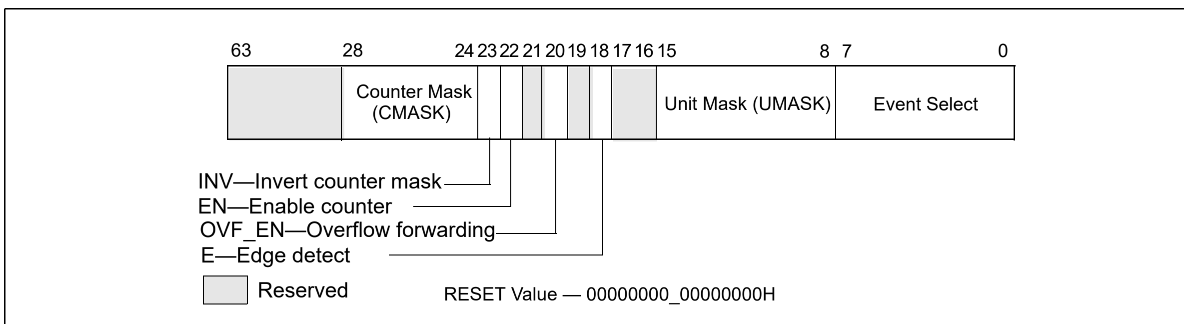


Figure 18-38. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see Table 19-17.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR_UNC_PERF_GLOBAL_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-39 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR_UNC_PERF_GLOBAL_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI_SEL_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

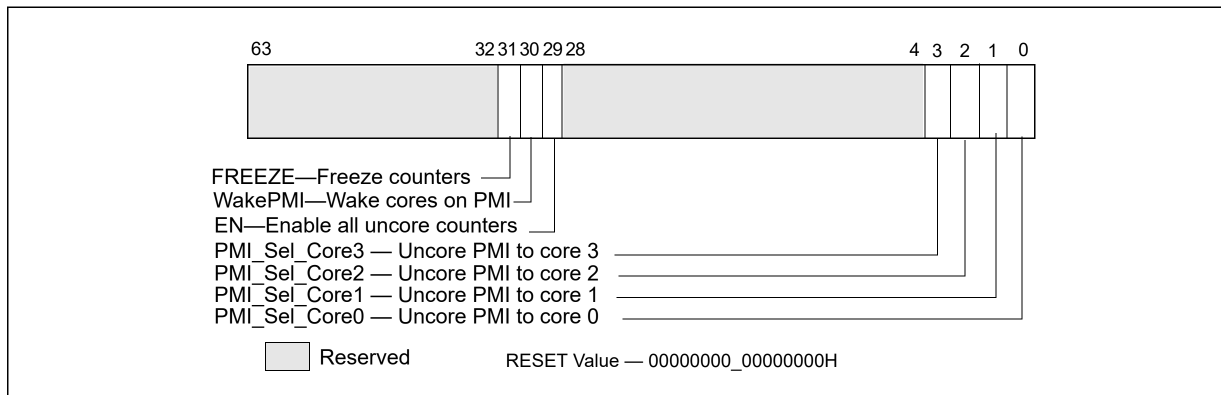


Figure 18-39. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-39 summarizes the number MSRs for uncore PMU for each box.

Table 18-39. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-19 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

18.9.6.1 Uncore Performance Monitoring Events

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

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

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

The C-Box uncore performance events described in Table 19-17 can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

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

18.9.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Intel microarchitecture code name Sandy Bridge-E. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 18.9.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 18.9 through Section 18.9.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 18-37 applies to Intel Core Processors with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2AH; Intel Xeon processor with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2DH supports an additional field for remote DRAM controller shown in Table 18-40. Additionally, the are some small differences in the non-architectural performance monitoring events (see Table 19-15).

Table 18-40. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Remote	30:23	(R/W): Remote DRAM Controller (either all 0s or all 1s)

18.9.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 18-41 summarizes the uncore PMU facilities providing MSR interfaces.

Table 18-41. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in “Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-22.

18.10 3RD GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 18.9 through Section 18.9.5. The non-architectural performance monitoring events supported by the processor core are listed in Table 19-15.

18.10.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in "Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-26.

18.11 4TH GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU's capability is similar to those described in Section 18.9 through Section 18.9.5, with some differences and enhancements summarized in Table 18-42. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.11.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Table 18-42. Core PMU Comparison

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-32 and Section 18.11.5.1.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.9.4.2.	See Section 18.9.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 18.9.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 17.10.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for Perfmon	See Section 18.11.5.	No	

18.11.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Intel microarchitecture code name Sandy Bridge, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Sandy Bridge is summarized in Table 18-43.

Table 18-43. PEBS Facility Comparison

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-21	Figure 18-35	
PEBS record layout	Table 18-44; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 18-23; enhanced fields at offsets 98H, A0H, A8H.	
Precise Events	See Table 18-32.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-33.	Table 18-33	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 18.9.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

18.11.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 18-44. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-44. PEBS Record Format for 4th Generation Intel Core Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 18.11.5.1)

The layout of PEBS records are almost identical to those shown in Table 18-23. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.9.4.2), PDIR (Section 18.9.4.4), and the equivalent capability of precise store in prior generation (see Section 18.11.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

18.11.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

Table 18-45. Precise Events That Supports Data Linear Address Profiling

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS

Table 18-45. Precise Events That Supports Data Linear Address Profiling (Contd.)

Event Name	Event Name
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32_PMC0-IA32_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program the an event listed in Table 18-45 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEBS_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-46.

Table 18-46. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES ▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 18-45.
Reserved	A8H	Always zero.

18.11.3.1 EventingIP Record

The PEBS record layout for processors based on Intel microarchitecture code name Haswell adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

18.11.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.9.5. The event codes are listed in Table 18-35. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-47.
- Supplier information (bits 30:16): see Table 18-48.
- Snoop response information (bits 37:31): see Table 18-38.

Table 18-47. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell microarchitecture)

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
COREWB	3	(R/W). Counts the number of modified cachelines written back.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	(R/W). Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	(R/W). Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_LOCK	10	(R/W). Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	(R/W). Counts the number of streaming store requests electronically.
Reserved	12-14	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-48. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 18-48. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_3CH, 06_46H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

Table 18-49. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	(R/W). L4 Cache
	Reserved	30:26	Reserved

18.11.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 18-48 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).

Table 18-50. MSR_OFFCORE_RSP_x Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	L3_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	(R/W). Hop 0 Remote supplier
	L3_MISS_REMOTE_HOP1	28	(R/W). Hop 1 Remote supplier
	L3_MISS_REMOTE_HOP2P	29	(R/W). Hop 2 or more Remote supplier
	Reserved	30	Reserved

18.11.5 Performance Monitoring and Intel® TSX

Chapter 16 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes the details of Intel® Transactional Synchronization Extensions (Intel TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances it's IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 18-40. The two additional bit fields are:

- **IN_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32_PERFEVTSEL2.

When the IA32_PERFEVTSELx MSR is programmed with both IN_TX=0 and IN_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN_TXCP bit. see Table 2-27.

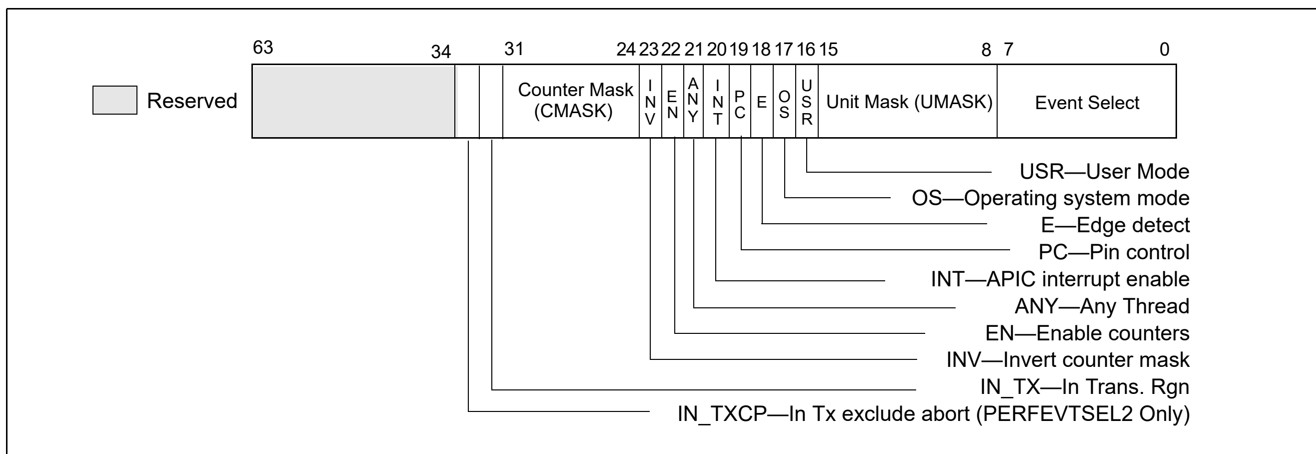


Figure 18-40. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32_PERFEVTSELx.IN_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32_PERFEVTSELx but occurring outside a transactional region are discarded. The following example illustrates using three counters to drill down cycles spent inside and outside of transactional regions:

- Program IA32_PERFEVTSEL2 to count Unhalted_Core_Cycles with (IN_TXCP=1, IN_TX=0), such that IA32_PMC2 will count cycles spent due to aborted TSX transactions;
- Program IA32_PERFEVTSEL0 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=1), such that IA32_PMC0 will count cycles spent by the transactional code regions;
- Program IA32_PERFEVTSEL1 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=0), such that IA32_PMC1 will count total cycles spent by the non-transactional code and transactional code regions.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they are listed in Table 19-9.

18.11.5.1 Intel TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events in Table 19-9 also support using PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORT ED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 18-51.

Table 18-51. TX Abort Information Field Definition

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
Reserved	63:40	Reserved

18.11.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 18.9.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 18-38.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-39 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-39 summarizes the number MSRs for uncore PMU for each box.

Table 18-52. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-19 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units are listed in Table 19-10.

18.11.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-31.

18.12 5TH GENERATION INTEL® CORE™ PROCESSOR AND INTEL® CORE™ M PROCESSOR PERFORMANCE MONITORING FACILITY

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU has the same capability as those described in Section 18.11. IA32_PERF_GLOBAL_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.

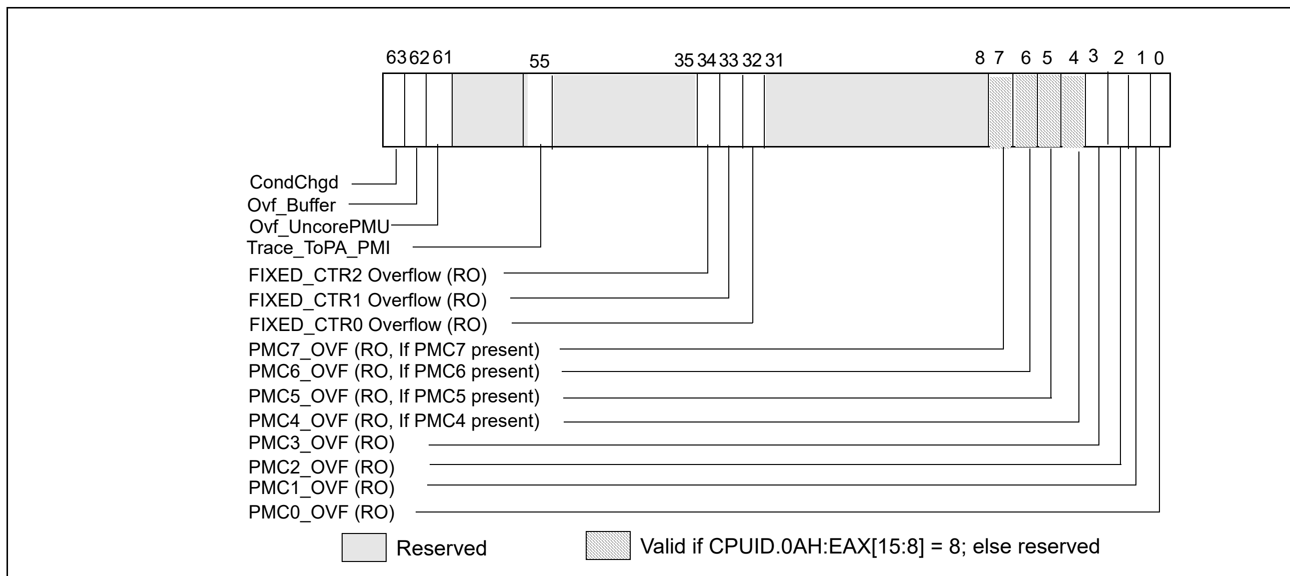


Figure 18-41. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

Details of Intel Processor Trace is described in Chapter 35, “Intel® Processor Trace”.
IA32_PERF_GLOBAL_OVF_CTRL MSR provide a corresponding reset control bit.

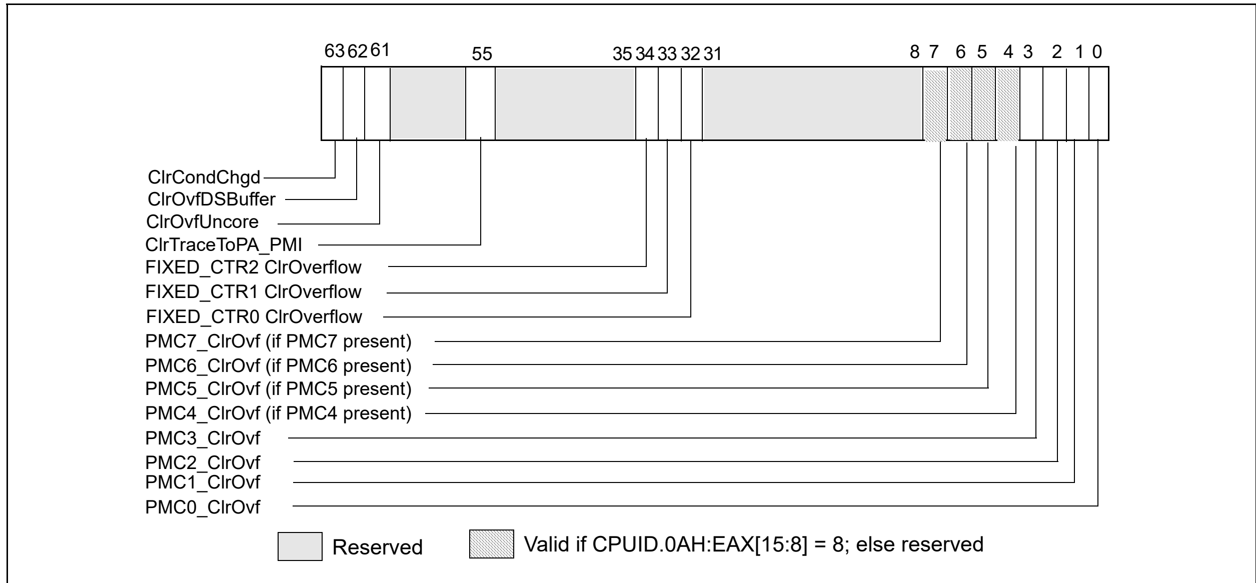


Figure 18-42. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, “Performance Monitoring Events”.

18.13 6TH GENERATION INTEL® CORE™ PROCESSOR AND 7TH GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU’s capability is similar to those described in Section 18.9 through Section 18.9.5, with some differences and enhancements summarized in Table 18-42. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.11.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 42, “Enclave Code Debug and Profiling”.

Table 18-53. Core PMU Comparison

Box	Intel® microarchitecture code name Skylake and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	CPUID enumerates # of counters.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_on_LBR with streamlined semantics. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-56.	See Table 18-32.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.13.1.4;	No	
LBR Record Format Encoding	000101b	000100b	Section 17.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 17.11
LBR Timing	Yes	No	Section 17.11.1
Call Stack Profiling	Yes, see Section 17.10	Yes, see Section 17.10	Use LBR facility
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 18.11.5;	See Section 18.11.5;	

18.13.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th and 7th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-54.

Table 18-54. PEBS Facility Comparison

Box	Intel® microarchitecture code name Skylake and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-21	Figure 18-21	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 18-55; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 18-44; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 18-56.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.9.4.2.	See Section 18.9.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTES

Precise events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

18.13.1.1 PEBS Data Format

The PEBS record format for the 6th and 7th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 18-55. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-55. PEBS Record Format for 6th Generation Intel Core Processor and 7th Generation Intel Core Processor Families

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.11.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-44.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.9.4.2), PDIR (Section 18.9.4.4), and data address profiling (Section 18.11.3).

In the core PMU of the 6th and 7th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the "applicable counter" field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS over-flow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

18.13.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake and Kaby Lake microarchitectures is shown in Table 18-56.

Table 18-56. Precise Events for the Skylake and Kaby Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST[1]	01H
		ALL_CYCLES[2]	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable[3]>	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED[2]	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED[4]	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED[2]	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRED.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBS_FRONTEND, see Section 18.13.2
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

18.13.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th and 7th generation Intel Core processors is largely unchanged from prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-46.

Table 18-57. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES. ▪ Other bits are zero.
Reserved	A8H	Always zero.

18.13.1.4 PEBS Facility for Front End Events

In the 6th and 7th generation Intel Core processors, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The frontend microarchitectural conditions supported by PEBS requires the following interfaces:

- The IA32_PERFEVTSELx MSR must select "FrontEnd_Retired" (C6H) in the EventSelect field (bits 7:0) and umask = 01H,
- The "FRONTEND_RETIRED" event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 18-58.
- Program the PEBS_EN_PMCx field of IA32_PEBS_ENABLE MSR as required.

Note the AnyThread field of IA32_PERFEVTSELx is ignored by the processor for the "FRONTEND_RETIRED" event.

The sub-event encodings supported by MSR_PEBS_FRONTEND.EVTSEL is given in Table 18-58.

Table 18-58. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL

Sub-Event Name	EVTSEL	Description
DSB_MISS	11H	Retired Instructions which experienced decode stream buffer (DSB) miss.
L1I_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss[1]. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.

NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 18-59.

Table 18-59. MSR_PEBS_FRONTEND Layout

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 18-58.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

18.13.1.5 FRONTEND_RETIRED

The FRONTEND_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) Frontend events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a frontend (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

18.13.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.9.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-60.
- Supplier information (bits 30:16): see Table 18-61.
- Snoop response information (bits 37:31): see Table 18-62.

Table 18-60. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake and Kaby Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	6:3	Reserved
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC prefetches into L3.

Table 18-60. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake and Kaby Lake Microarchitectures)

Bit Name	Offset	Description
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	10:9	Reserved
STRM_ST	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-61 lists the supplier information field that applies to 6th and 7th generation Intel Core processors. (6th generation Intel Core processor CPUID signature: 06_4EH, 06_5EH; 7th generation Intel Core processor CPUID signature: 06_8EH, 06_9EH).

Table 18-61. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_4EH, 06_5EH and 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available.
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT	22	(R/W). L4 Cache (if L4 is present in the processor)
	Reserved	25:23	Reserved
	DRAM	26	(R/W). Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	(R/W). L4 cache super line hit (if L4 is present in the processor)

Table 18-62 lists the snoop information field that apply to processors with CPUID signature 06_4EH, 06_5EH and 06_8EH, 06_9E.

Table 18-62. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signature 06_4EH, 06_5EH and 06_8EH, 06_9E)

Subtype	Bit Name	Offset	Description
Snoop Info	SNOOP_NONE	31	(R/W). No details on snoop-related information
	SNOOP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNOOP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

18.14 INTEL® XEON PHI™ PROCESSOR 7200/5200/3200 PERFORMANCE MONITORING

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel® Atom™ processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3 in the SDM. The processor supports AnyThread counting in three architectural performance monitoring events.

18.14.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32_FIXED_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32_PERFEVTSELx.
- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor untile changes.

18.14.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32_PMC0 (see also Section 17.4.9, “BTS and DS Save Area”).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4).

The list of PEBS events supported in the processor is shown in the following table.

Table 18-63. PEBS Performance Events for the Knights Landing Microarchitecture

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		CALL	F9H	No
		REL_CALL	FDH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		FAR_BRANCH	BFH	No
		RETURN	F7H	No

Table 18-63. PEBS Performance Events for the Knights Landing Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		RETURN	F7H	No
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H	Yes
		L2_MISS_LOADS	04H	Yes
		DLTB_MISS_LOADS	08H	Yes
RECYCLEQ	03H	LD_BLOCK_ST_FORWARD	01H	Yes
		LD_SPLITS	08H	Yes

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 18-64, and each field in the PEBS record is 64 bits long.

Table 18-64. PEBS Record Format for the Knights Landing Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	PSDLA
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

18.14.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-65 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-65. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 18-66. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.6.3 for details.

Table 18-66. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		15	ANY_REQUEST	Account for any requests.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.

Table 18-66. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
		22	MCDRAM_FAR_OR_L2_HIT_OTHER_TILE_FAR	MCDRAM Far or Other tile L2 hit far.
		23	DRAM_NEAR	DRAM Local.
		24	DRAM_FAR	DRAM Far.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMC0. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMC0. This bit is reserved for OFFCORE_RESP_1 event.

18.14.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR_OFFCORE_RSP0.[bit 38] with the choice of request type specified in MSR_OFFCORE_RSP0.[bit 15:0].

Refer to Section 18.6.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR_OFFCORE_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

18.15 PERFORMANCE MONITORING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32_MISC_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32_DS_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR_PEBS_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 18-67 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events are listed in Chapter 19, “Performance-Monitoring Events.”

Table 18-67. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H

Table 18-67. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH

Table 18-67. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH

NOTES:

1. MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events (see Table 19-29) are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events (see Table 19-30) are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

  The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting** — A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling** — A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts

the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.

- **Processor event-based sampling (PEBS)** — In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

18.15.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-67) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-67) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 18-43 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- **USR flag, bit 2** — When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- **OS flag, bit 3** — When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)

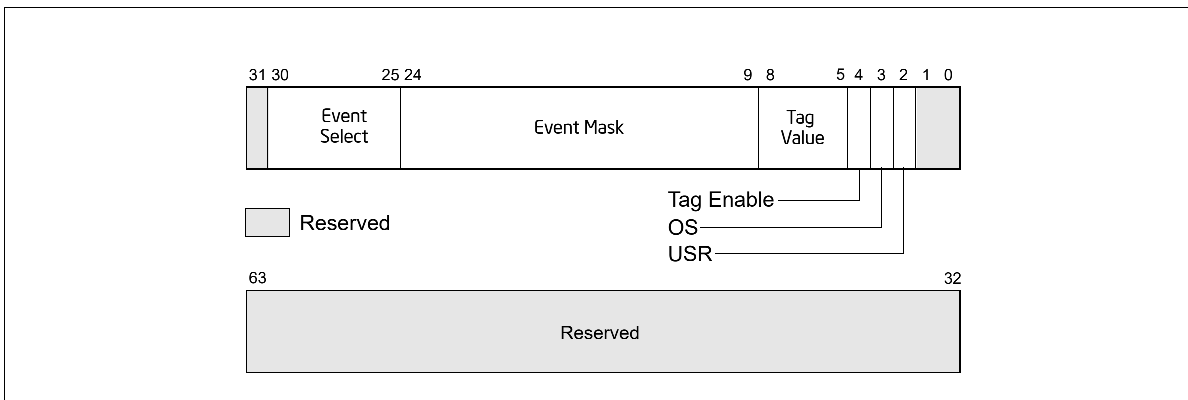


Figure 18-43. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support

- **Tag enable, bit 4** — When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.15.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30)** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and

USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-67 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

18.15.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 18-67). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR_BPU_COUNTER0 and MSR_BPU_COUNTER1.
  - MSR_BPU_COUNTER2 and MSR_BPU_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR_MS_COUNTER0 and MSR_MS_COUNTER1.
  - MSR_MS_COUNTER2 and MSR_MS_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR_FLAME_COUNTER0 and MSR_FLAME_COUNTER1.
  - MSR_FLAME_COUNTER2 and MSR_FLAME_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR_IQ_COUNTER0 and MSR_IQ_COUNTER1.
  - MSR_IQ_COUNTER2 and MSR_IQ_COUNTER3.
  - MSR_IQ_COUNTER4 and MSR_IQ_COUNTER5.

The MSR_IQ_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR_BPU_COUNTER0 can start MSR_BPU_COUNTER2 and vice versa, and MSR_BPU_COUNTER1 can start MSR_BPU_COUNTER3 and vice versa (see Section 18.15.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 18-44). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

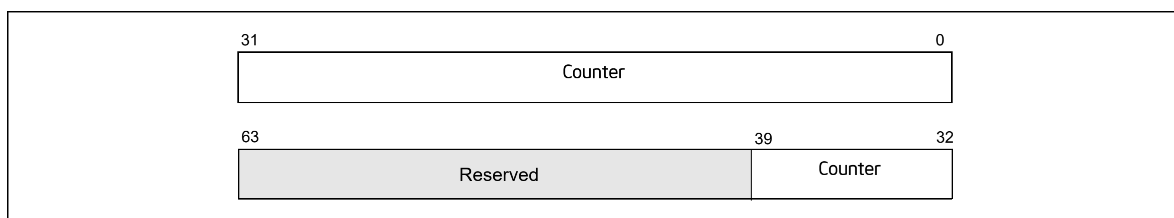


Figure 18-44. Performance Counter (Pentium 4 and Intel Xeon Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

18.15.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 18-67). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 18-45 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag**, **bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field**, **bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag**, **bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag**, **bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.15.5.2, “Filtering Events”). The complement flag is not active unless the compare flag is set.
- **Threshold field**, **bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.15.5.2, “Filtering Events”).
- **Edge flag**, **bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

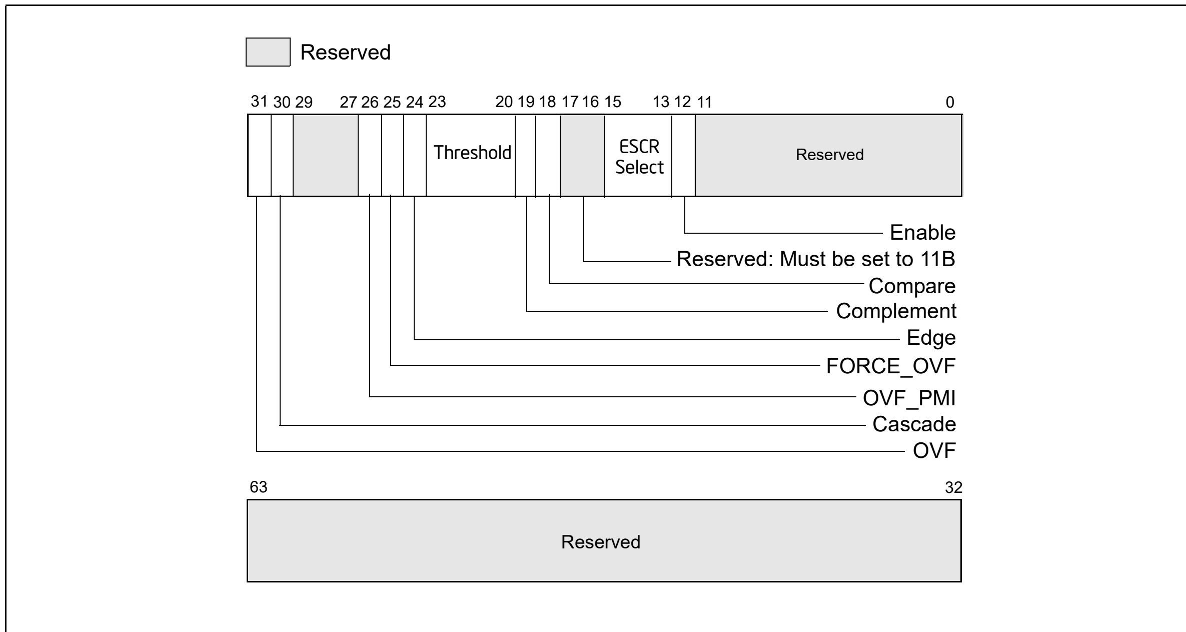


Figure 18-45. Counter Configuration Control Register (CCCR)

- **FORCE_OVF flag**, **bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI flag**, **bit 26** — When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag**, **bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.15.2, "Performance Counters," for further details); when clear, disables cascading of counters.
- **OVF flag**, **bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.
2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one "stage" forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 18.15.5, "Programming the Performance Counters for Non-Retirement Events."

18.15.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 17.4.5, "Branch Trace Store (BTS)," and Section 18.15.7, "Processor Event-Based Sampling (PEBS)," for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 17.4.9, "BTS and DS Save Area."

18.15.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event using the values in the ESCR restrictions row in Table 19-29, Chapter 19.
3. Match the CCCR Select value and ESCR name in Table 19-29 to a value listed in Table 18-67; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which the are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

18.15.5.1 Selecting Events to Count

Table 19-30 in Chapter 19 lists a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event listed in Table 19-30, setup information is provided. Table 18-68 gives an example of one of the events.

Table 18-68. Event Example

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit 0: MMNP	Branch Not-taken Predicted
		1: MMNM	Branch Not-taken Mispredicted
		2: MMTP	Branch Taken Predicted
		3: MMTM	Branch Taken Mispredicted

Table 18-68. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

For Table 19-29 and Table 19-30, Chapter 19, the name of the event is listed in the Event Name column and parameters that define the event and other information are listed in the Event Parameters column. The Parameter Value and Description columns give specific parameters for the event and additional description information. Entries in the Event Parameters column are described below.

- **ESCR restrictions** — Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR** — Lists which performance counters are associated with each ESCR. Table 18-67 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select** — Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask** — Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select** — Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 18-67.
- **Event specific notes** — Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS** — Indicates if PEBS is supported for the event (only supplied for at-retirement events listed in Table 19-30.)
- **Requires additional MSR for tagging** — Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events listed in Table 19-30.)

NOTE

The performance-monitoring events listed in Chapter 19, "Performance-Monitoring Events," are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

Using information in Table 19-29, Chapter 19, an event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 18-67.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.

6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

NOTE

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 18.15.5.2, “Filtering Events.”

18.15.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 3 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a “greater than” or a “less than or equal to” comparison of the input value vs. a threshold value can be made. The complement flag selects “less than or equal to” (flag set) or “greater than” (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a “rising edge” event; that is, a false-to-true transition. Figure 18-46 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 18.15.5.1, “Selecting Events to Count.”

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
   — Set the compare flag.
   — Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
   — Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 18.15.5.3, “Starting Event Counting.”

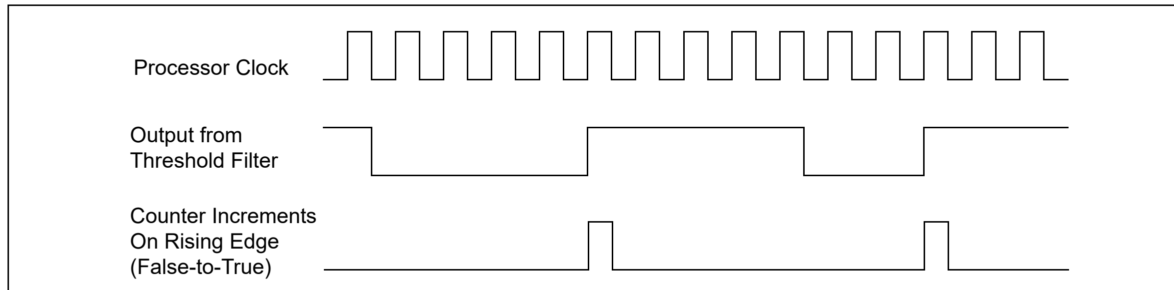


Figure 18-46. Effects of Edge Filtering

18.15.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 18.15.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 18.15.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 18.15.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 18.15.5.6, "Cascading Counters").

18.15.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 18.15.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 18.15.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-67 used as an operand.

This setup procedure is continued in the next section, Section 18.15.5.5, "Halting Event Counting."

18.15.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 18.15.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

18.15.5.6 Cascading Counters

As described in Section 18.15.2, "Performance Counters," eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 18-67). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR_IQ_CCCR0 through MSR_IQ_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 18-45 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR_MO B_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It's possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14 cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

Example 18-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 18.15.5.1, "Selecting Events to Count." Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 18.15.5.8, "Generating an Interrupt on Overflow."

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

18.15.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily_DisplayModel 0F_02, 0F_03, 0F_04, 0F_06. This feature uses bit 11 in CCCRs associated with the IQ block. See Table 18-69.

Table 18-69. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1|2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INTO0	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INTO3	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INTO4	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INTO5	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 18-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

18.15.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 18.15.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

18.15.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no_event, which generally has a select value of 0).

18.15.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

Tables 19-30 through 19-34 list predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire** — In at-retirement event descriptions, the term "bogus" refers to instructions or μops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or μops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction_Retired and Uops_Retired in Table 19-30) can count instructions or μops that are retired based on the characterization of bogus" versus non-bogus.

- **Tagging —** Tagging is a means of marking μops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per μop and a direct count of the event would not provide an indication of how many μops encountered that event.

  The tagging mechanisms allow a μop to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per μop, rather than once per event. For example, a μop may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired μops that encountered a cache miss) will increment only once for that μop. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, "Related Literature").
- **Replay —** To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist —** When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μops before they begin and are costly.

18.15.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-30, a μop may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count μops tagged with that mechanism:

- **Front-end tagging —** This mechanism pertains to the tagging of μops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event
- **Execution tagging —** This mechanism pertains to the tagging of μops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.
- **Replay tagging —** This mechanism pertains to tagging of μops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.
- **No tags —** This mechanism does not use tags. It uses the Instr_retired and the Uops_ retired events.

Each tagging mechanism is independent from all others; that is, a μop that has been tagged using one mechanism will not be detected with another mechanism's tagged-μop detector. For example, if μops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged μops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of μops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

Table 19-30 lists the performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag μop and count tagged μops.

18.15.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts μops that have been tagged as encountering any of the following events:

- **μop decode events** — Tagging μops for μop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.
- **Trace cache events** — Tagging μops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR (see Table 19-32).

Table 19-30 describes the Front_end_event and Table 19-32 describes metrics that are used to set up a Front_end_event count.

The MSRs specified in the Table 19-30 that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front_end_event event mask must be set to count events. None of the events currently supported requires the use of the MSR_TC_PRECISE_EVENT MSR.

18.15.6.3 Tagging Mechanism For Execution_event

Table 19-30 describes the Execution_event and Table 19-33 describes metrics that are used to set up an Execution_event count.

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect μops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular μop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged μops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. This mechanism is summarized in the Table 19-33 metrics that are supported by the execution tagging mechanism. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 18.15.7.3, "Setting Up the PEBS Buffer.") It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

18.15.6.4 Tagging Mechanism for Replay_event

Table 19-30 describes the Replay_event and Table 19-34 describes metrics that are used to set up an Replay_event count.

The replay mechanism enables tagging of μops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μop that may experience the replay in the MSR_PEBS_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBS_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBS_ENABLE MSR.

The Table 19-34 lists the metrics that are support the replay tagging mechanism and the at-retirement events that use the replay tagging mechanism, and specifies how the appropriate MSRs need to be configured. The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 17.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE_MSR. Each of these metrics requires that the Replay_Event (see Table 19-30) be used to count the tagged μops.

18.15.7 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 17.4.9, "BTS and DS Save Area"). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can only be carried out using the one performance counter, the MSR_IQ_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32_PMC0 and IA32_PERFEVTSEL0 MSRs (See Section 18.4.4).

18.15.7.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR_PEBS_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR_PEBS_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

18.15.7.2 Setting Up the DS Save Area

Section 17.4.9.2, "Setting Up the DS Save Area," describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

18.15.7.3 Setting Up the PEBS Buffer

Only the MSR_IQ_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 17-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR_PEBS_ENABLE MSR.
3. Set up the MSR_IQ_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS as described in Tables 19-30 through 19-34.

18.15.7.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.5, "Writing the DS Interrupt Service Routine," for guidelines for writing the DS ISR.

18.15.7.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

18.15.8 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

18.16 PERFORMANCE MONITORING AND INTEL HYPER-THREADING TECHNOLOGY IN PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 18.15. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_TC_PRECISE_EVENT.

18.16.1 ESCR MSRs

Figure 18-47 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1_USR flag**, **bit 0** — When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.

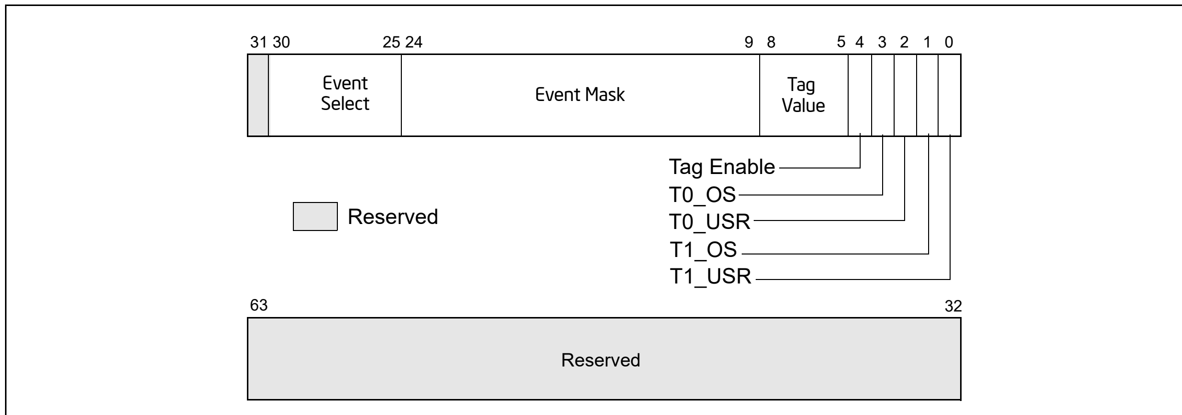


Figure 18-47. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology

- **T1_OS flag**, **bit 1 —** When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1_OS and T1_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0_USR flag**, **bit 2 —** When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0_OS flag**, **bit 3 —** When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0_OS and T0_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable**, **bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.15.6, "At-Retirement Counting."
- **Tag value field**, **bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field**, **bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field**, **bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, "Identifying Logical Processors in an MP System," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.16.4, "Performance Monitoring Events"). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

18.16.2 CCCR MSRs

Figure 18-48 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag**, **bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field**, **bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field**, **bits 16 and 17 —** Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

00 — None. Count only when neither logical processor is active.

01 — Single. Count only when one logical processor is active (either 0 or 1).

10 — Both. Count only when both logical processors are active.

11 — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the "wait for SIPI" state is considered inactive.

- **Compare flag**, **bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

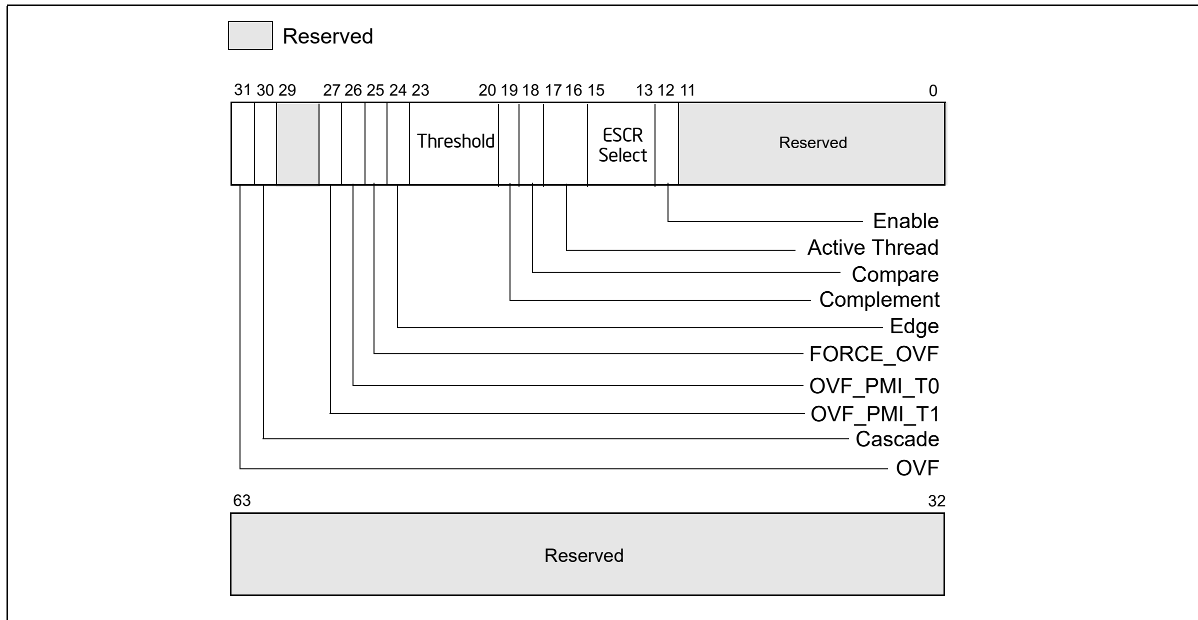


Figure 18-48. Counter Configuration Control Register (CCCR)

- **Complement flag**, **bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.15.5.2, "Filtering Events"). The compare flag is not active unless the compare flag is set.
- **Threshold field**, **bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.15.5.2, "Filtering Events").
- **Edge flag**, **bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE_OVF flag**, **bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI_T0 flag**, **bit 26** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF_PMI_T1 flag**, **bit 27** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.

- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.15.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

18.16.3 IA32_PEBS_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR_PEBS_ENABLE MSR: bit 25 (ENABLE_PEBS_MY_THR) and 26 (ENABLE_PEBS_OTH_THR) respectively. These bits do not explicitly identify a specific logical processor by logic processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running (“my thread”) or for the other logical processor in the physical package on which the agent is not running (“other thread”).

PEBS is supported for only a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can be carried out only with two performance counters: MSR_IQ_CCCR4 (MSR address 370H) for logical processor 0 and MSR_IQ_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE_PEBS_MY_THR and ENABLE_PEBS_OTH_THR bits in the MSR_PEBS_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

18.16.4 Performance Monitoring Events

All of the events listed in Table 19-29 and 19-30 are available in an Intel Xeon processor MP. When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS)** — The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI)** — The event cannot be qualified as being associated with a specific logical processor.

Table 19-35 gives logical processor specific information (TS or TI) for each of the events described in Tables 19-29 and 19-30. If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-70) depends only on the setting of the T0_USR and T0_OS flags in the ESCR being used to set up the event counter. The T1_USR and T1_OS flags have no effect on the count.

Table 18-70. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) or T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a)T0 in Os or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI in Chapter 19, the effect of selectively specifying T0_USR, T0_OS, T1_USR, T1_OS bits is shown in Table 18-71.

Table 18-71. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

18.17 COUNTING CLOCKS ON SYSTEMS WITH INTEL HYPER-THREADING TECHNOLOGY IN PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE

18.17.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the global_power_events and specify the RUNNING sub-event mask and the desired T0_OS/T0_USR/T1_OS/T1_USR bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

18.17.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than “no_event”; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to “1”.
4. Set the threshold to “15” and the complement to “1” in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g. by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical proces-

sors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

18.18 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor’s bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions
- Enhanced Intel SpeedStep Technology transitions (P-state transitions)

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 18.18.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock** — The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock** — This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock** — These clockticks increment at a fixed frequency and help coordinate the bus on some systems.
- **Core Crystal Clock** — This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks** — Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks** — Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter** — See Section 17.16, “Time-Stamp Counter”.
- **Reference Clockticks** — TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 17.16, "Time-Stamp Counter," for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI** — Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI** — Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

18.18.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and umask 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

18.18.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 14, "Power and Thermal Management"), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32_FIXED_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32_MPERF, and IA32_FIXED_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

18.18.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 "Information Returned by CPUID Instruction"), the nominal TSC frequency can be determined by using the following equation:

Nominal TSC frequency = (CPUID.15H.ECX[31:0] * CPUID.15H.EBX[31:0]) ÷ CPUID.15H.EAX[31:0]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 18-72 can be used to look up the nominal core crystal clock frequency.

Table 18-72. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series[1]	Nominal Core Crystal Clock Frequency
Future Intel® Xeon® processors with CPUID signature 06_55H	25 MHz
6th and 7th generation Intel® Core™ processors (does not include Intel® Xeon® processors)	24 MHz
Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors)	19.2 MHz

NOTES:

1. For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

18.18.3.1 For Intel® Processors Based on Microarchitecture Code Name Sandy Bridge, Ivy Bridge, Haswell and Broadwell

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

18.18.3.2 For Intel® Processors Based on Microarchitecture Code Name Nehalem

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

18.18.3.3 For Intel® Atom™ Processors Based on the Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The nominal TSC frequency can be obtained by multiplying the maximum resolved bus ratio, which can be read from MSR_PLATFORM_ID[13:8], by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR_FSB_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*.

18.18.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] at (0CDH), see Chapter 2, “Model-Specific Registers (MSRs)” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR_PLATFORM_ID[12:8]. It corresponds to the Processor Base frequency.
- IF XE operation is enabled, the maximum resolved bus ratio is given in MSR_PERF_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR_PERF_STATUS[31] is set, XE operation is enabled. The MSR_PERF_STATUS[31] field is read-only.

18.19 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 18-49, it provides enumeration of a variety of interfaces:

- IA32_PERF_CAPABILITIES.LBR_FMT[bits 5:0]: encodes the LBR format, details are described in Section 17.4.8.1.
- IA32_PERF_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist, see Section 18.4.4.2.
- IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers, see Section 18.4.4.2.
- IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records, see Section 18.4.4.2.
- IA32_PERF_CAPABILITIES.SMM_FRZ[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1, see Section 18.19.1.
- IA32_PERF_CAPABILITIES.FULL_WRITE[13]: Indicates the processor supports IA32_A_PMCx interface for updating bits 32 and above of IA32_PMCx, see Section 18.2.5.

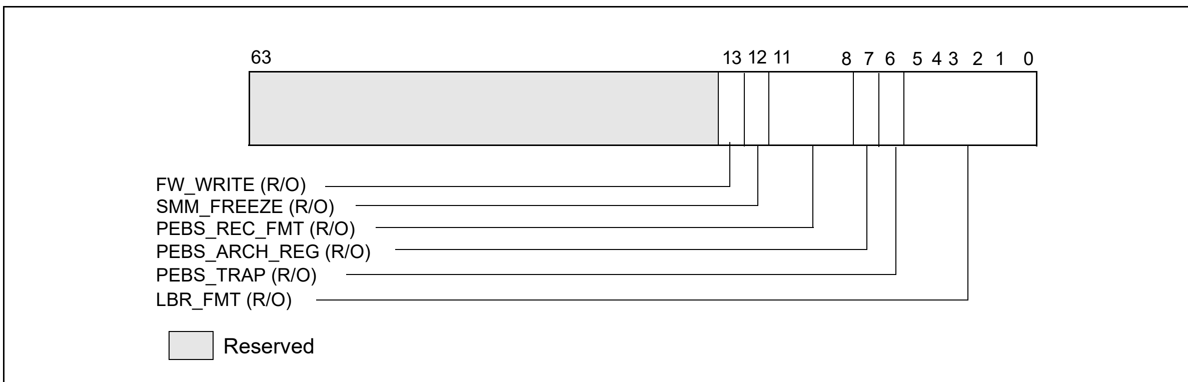


Figure 18-49. Layout of IA32_PERF_CAPABILITIES MSR

18.19.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 17.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)”) are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32_PERF_CAPABILITIES MSR. If IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE_WHILE_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored , after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

18.20 PERFORMANCE MONITORING AND DUAL-CORE TECHNOLOGY

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 18.16, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture").

18.21 PERFORMANCE MONITORING ON 64-BIT INTEL XEON PROCESSOR MP WITH UP TO 8-MBYTE L3 CACHE

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 18.1 and Section 18.16) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 18-50.

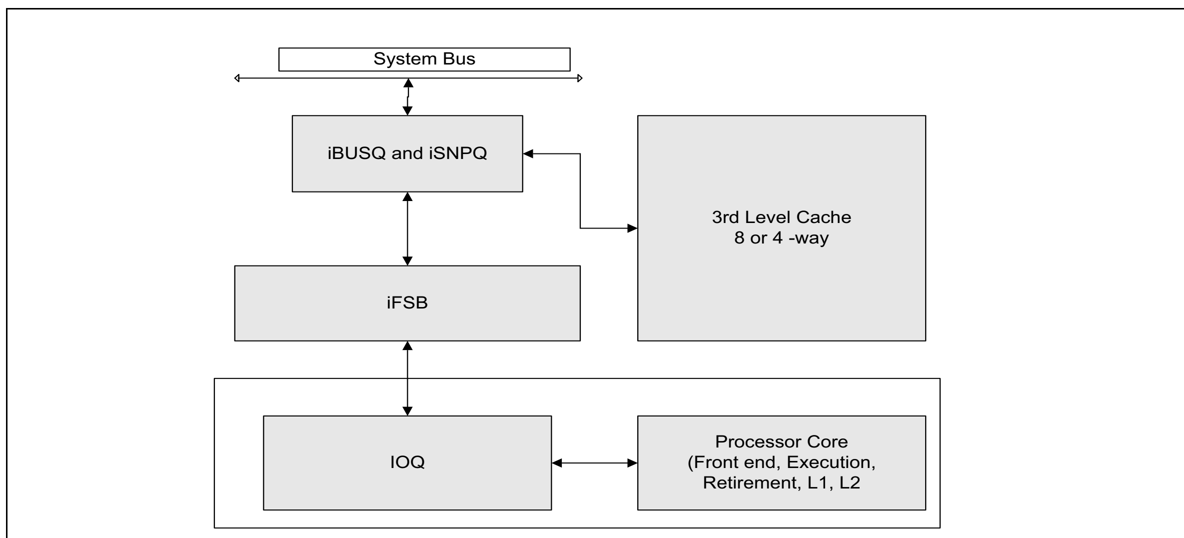


Figure 18-50. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- **IBUSQ event** — This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR_IFSB_IBUSQ0 and MSR_IFSB_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 18-51.

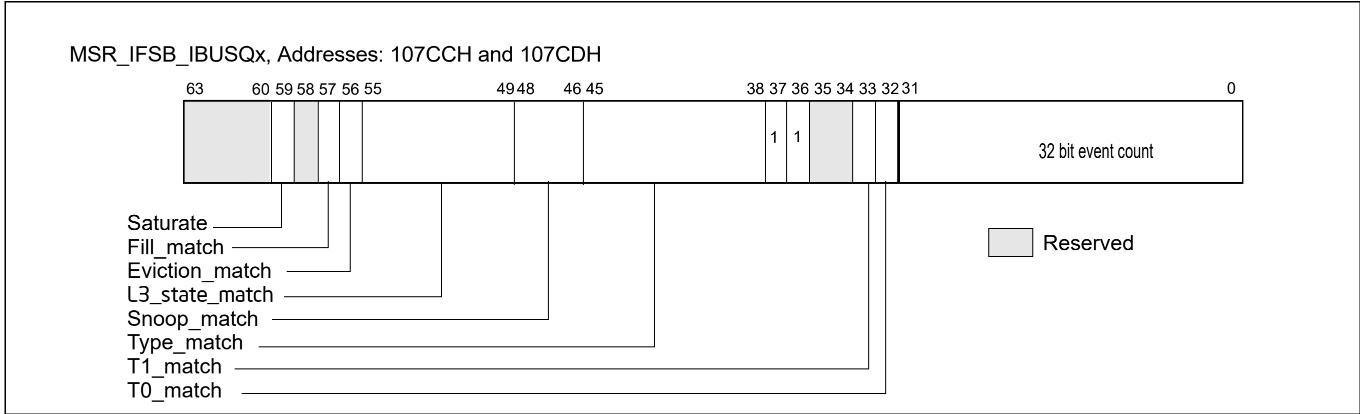


Figure 18-51. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- **ISNPQ event** — This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR_IFSB_ISNPQ0 and MSR_IFSB_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 18-52.

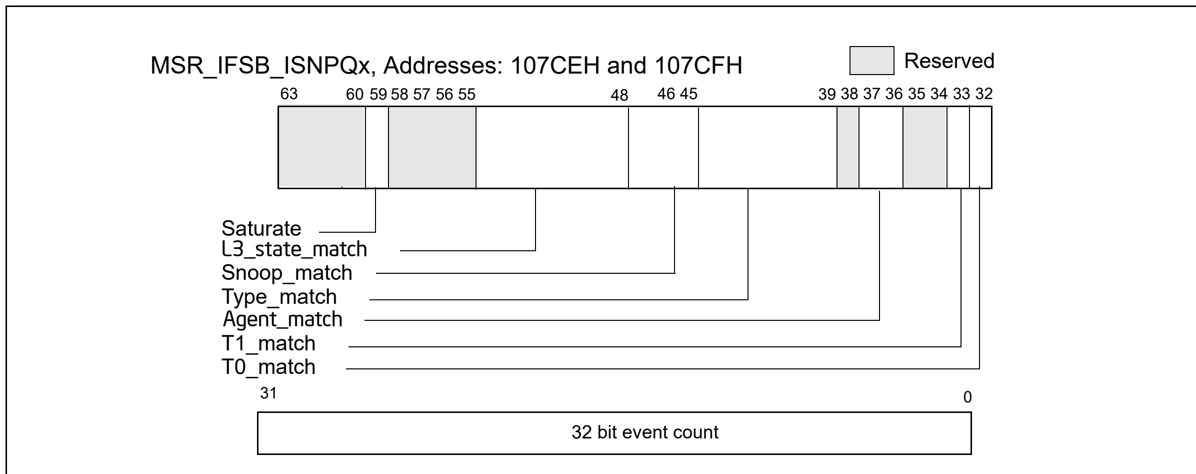


Figure 18-52. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- **EFSB event** — This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR_EFSB_DRDY0 and MSR_EFSB_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 18-53.

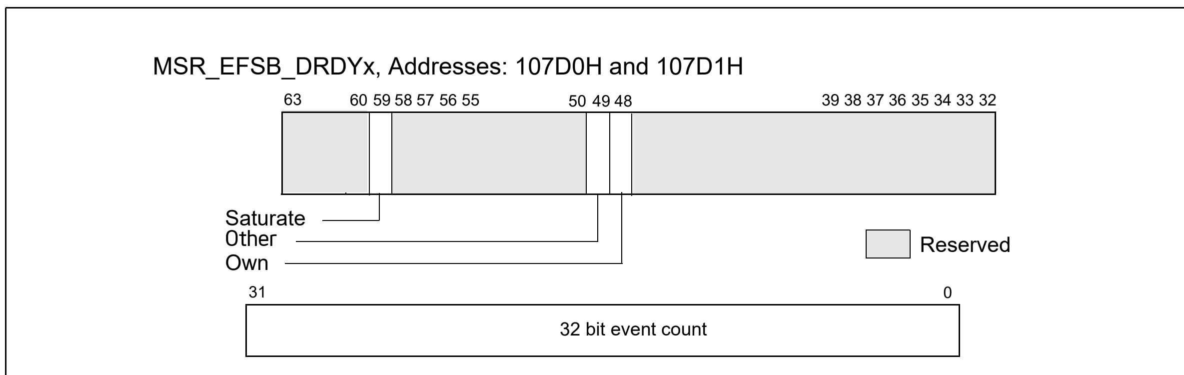


Figure 18-53. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- **IBUSQ Latency event** — This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR_IFSB_CTRL6[bit 26] to 1; the count freezes after software sets MSR_IFSB_CTRL6[bit 26] to 0. MSR_IFSB_CNTR7 acts as a 64-bit event counter for this event. See Figure 18-54.

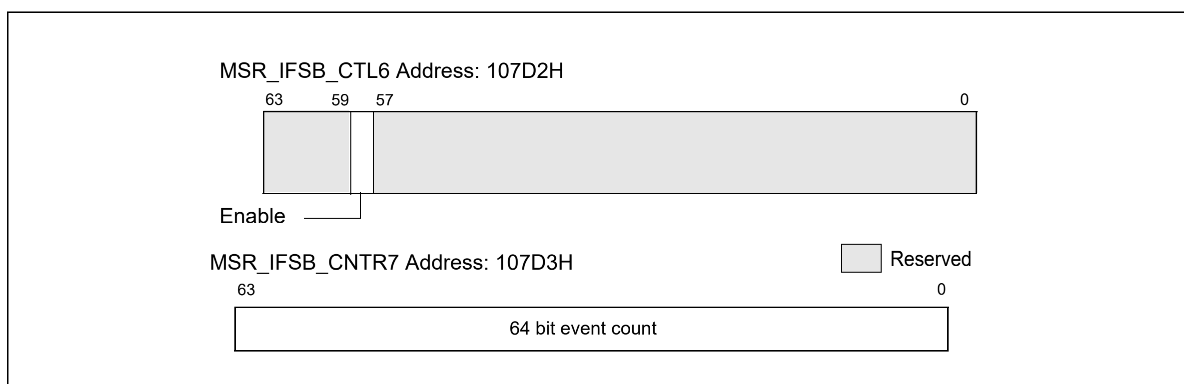


Figure 18-54. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H

18.22 PERFORMANCE MONITORING ON L3 AND CACHING BUS CONTROLLER SUB-SYSTEMS

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system

provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through additional control logic. See Figure 18-55 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-55 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.

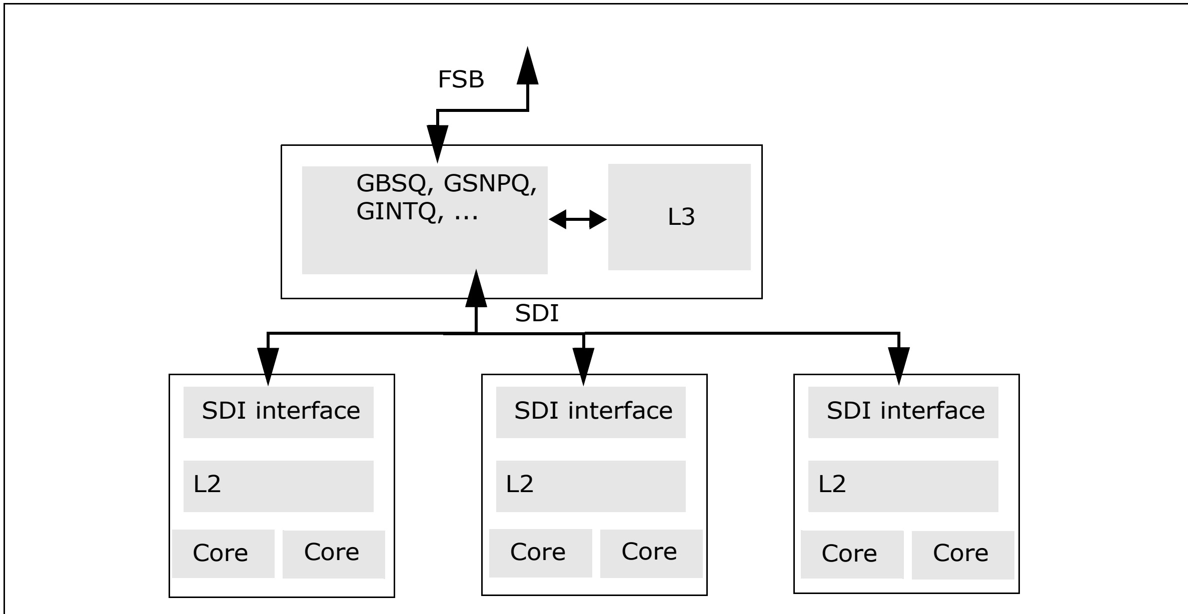


Figure 18-55. Block Diagram of Intel Xeon Processor 7400 Series

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.16) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

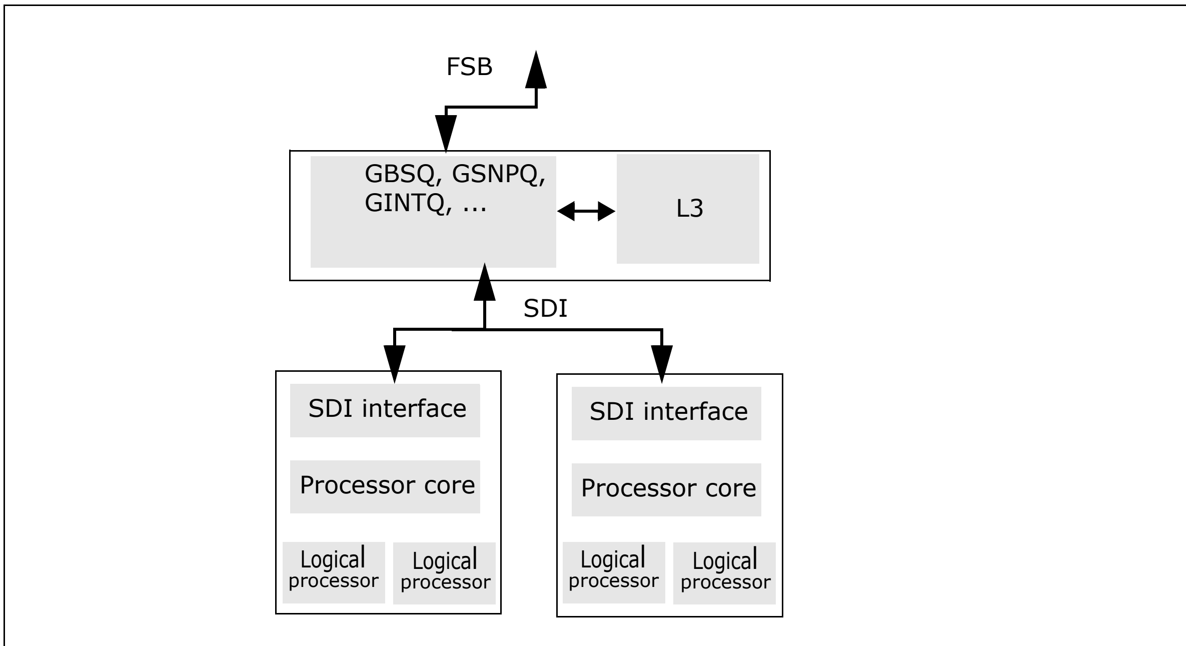


Figure 18-56. Block Diagram of Intel Xeon Processor 7100 Series

18.22.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR_EMON_L3_GL_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3) are dedicated to counting GSNPQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Four MSRs (MSR_EMON_L3_CTR_CTL4, MSR_EMON_L3_CTR_CTL5, MSR_EMON_L3_CTR_CTL6, and MSR_EMON_L3_CTR_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.

18.22.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 is given in Figure 18-57. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.

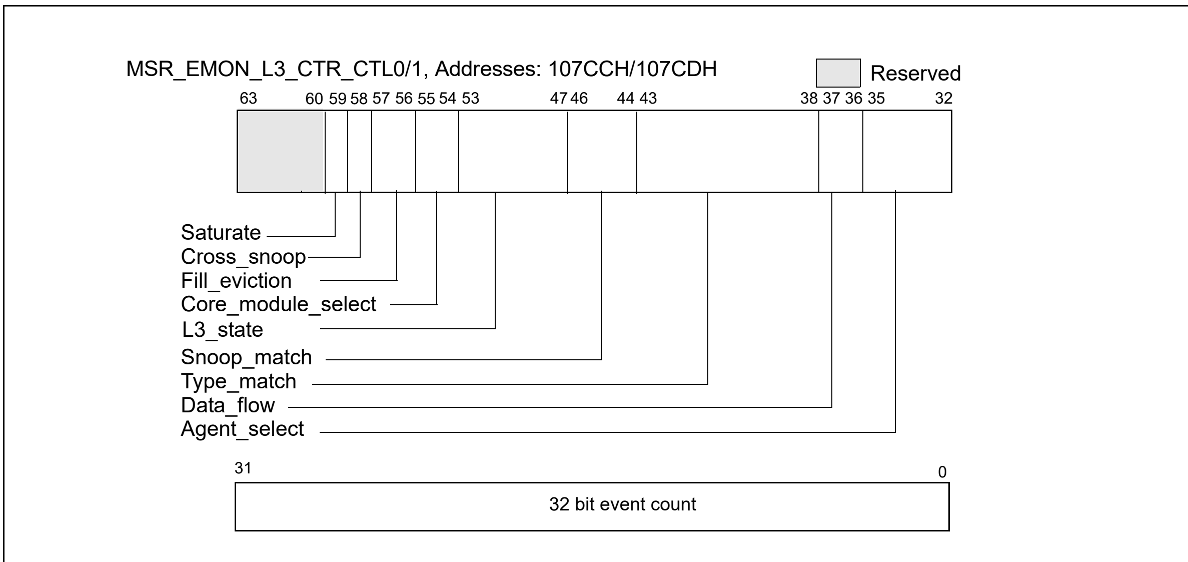


Figure 18-57. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- Data_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core_Module_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series,

  — 00B: Match transactions from any core in the physical package

  — 01B: Match transactions from this core only

  — 10B: Match transactions from the other core in the physical package

  — 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series,

  — 00B: Match transactions from any dual-core module in the physical package

  — 01B: Match transactions from this dual-core module only

  — 10B: Match transactions from either one of the other two dual-core modules in the physical package

- 11B: Match transaction from more than one dual-core modules in the physical package
- Fill_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross_Snoop (bit 58): The encodings are \
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

18.22.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 is given in Figure 18-58. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core_Module_Select (bits 56:54): Bit 56 enables Core_Module_Select matching. If bit 56 is clear, Core_Module_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one core (irrespective which core) in the physical package
  - 01B: Match transactions from this core and not the other core
  - 10B: Match transactions from the other core in the physical package, but not this core
  - 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
  - 01B: Match transactions from one or more dual-core modules.
  - 10B: Match transactions from two or more dual-core modules.
  - 11B: Match transaction from all three dual-core modules in the physical package.
- Block_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

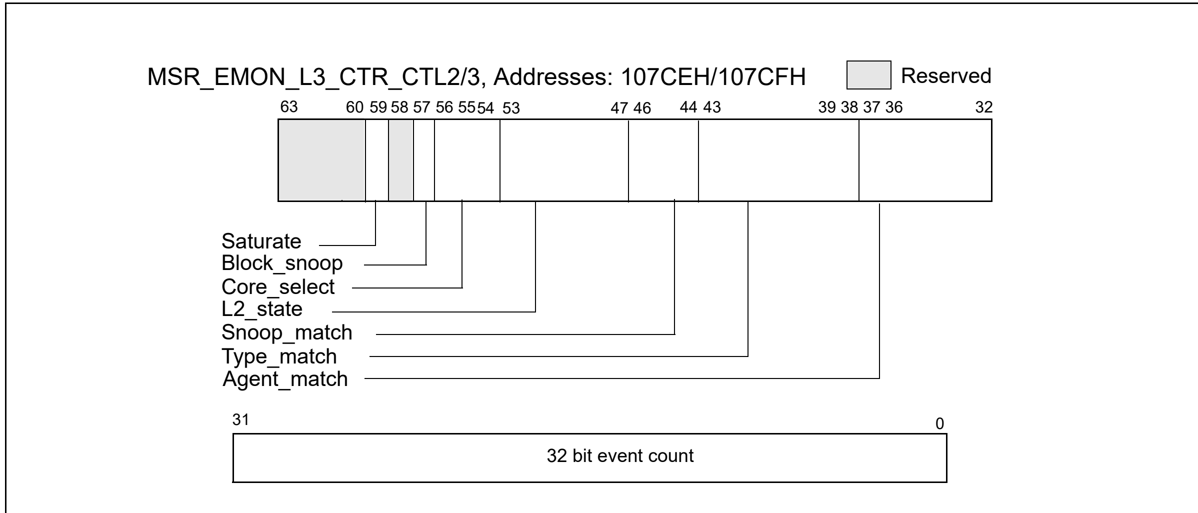


Figure 18-58. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

18.22.4 FSB Event Interface

The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 18-59. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

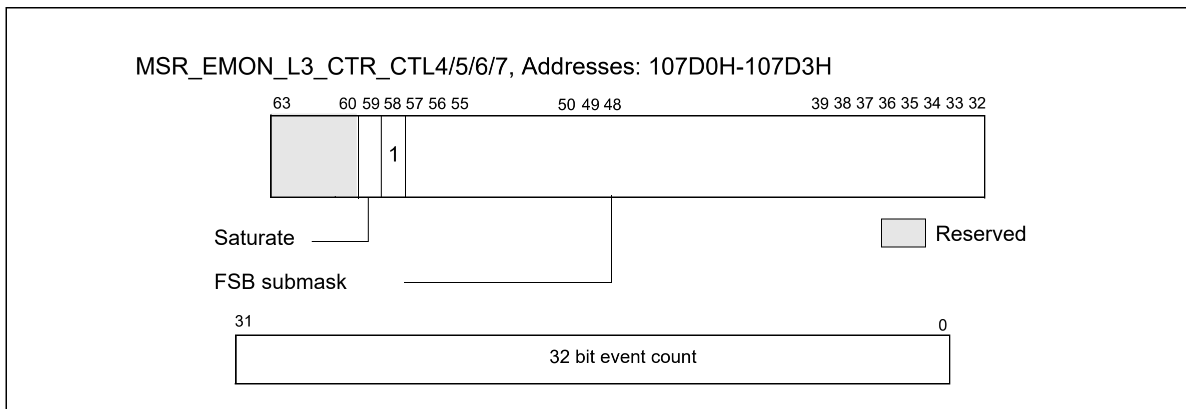


Figure 18-59. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

18.22.4.1 FSB Sub-Event Mask Interface

- FSB_type (bit 37:32): Specifies different FSB transaction types originated from this physical package
- FSB_L_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package
- FSB_L_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package

- FSB_L_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package
- FSB_L_defer (bit 41): Count DEFER responses to this processor’s transactions
- FSB_L_retry (bit 42): Count RETRY responses to this processor’s transactions
- FSB_L_snoop_stall (bit 43): Count snoop stalls to this processor’s transactions
- FSB_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY)
- FSB_DRDY (bit 45): Count DRDY assertions by this processor
- FSB_BNR (bit 46): Count BNR assertions by this processor
- FSB_IOQ_empty (bit 47): Counts each bus clocks when the IOQ is empty
- FSB_IOQ_full (bit 48): Counts each bus clocks when the IOQ is full
- FSB_IOQ_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ
- FSB_WW_data (bit 50): Counts back-to-back write transaction’s data phase.
- FSB_WW_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB_WR_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB_RW_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB_other_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY)
- FSB_other_DRDY (bit 55): Count DRDY assertions by another agent
- FSB_other_snoop_stall (bit 56): Count snoop stalls on the FSB due to another agent
- FSB_other_BNR (bit 57): Count BNR assertions from another agent

18.22.5 Common Event Control Interface

The MSR_EMON_L3_GL_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL_freeze_cmd (bit 0): Freeze the event counters specified by the GL_event_select field.
- GL_unfreeze_cmd (bit 1): Unfreeze the event counters specified by the GL_event_select field.
- GL_reset_cmd (bit 2): Clear the event count field of the event counters specified by the GL_event_select field. The event select field is not affected.
- GL_event_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR_EMON_L3_CTR_CTL0, bit 23 corresponds to MSR_EMON_L3_CTR_CTL7.
- GL_event_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR_EMON_L3_CTR_CTL0, bit 55 corresponds to MSR_EMON_L3_CTR_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 18-57 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

18.23 PERFORMANCE MONITORING (P6 FAMILY PROCESSOR)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

Table 19-38, Chapter 19, lists the events that can be counted with the P6 family performance monitoring counters.

NOTE

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

NOTE

The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed in Table 19-38 are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

18.23.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 18-60 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7)** — Selects the event logic unit to detect certain microarchitectural conditions (see Table 19-38, for a list of events and their 8-bit codes).
- **Unit mask (UMASK) field (bits 8 through 15)** — Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states (see Table 19-38).

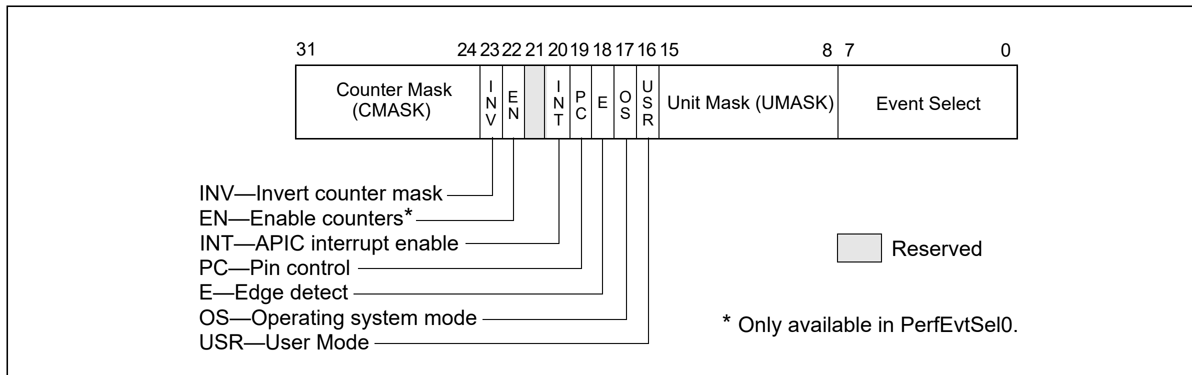


Figure 18-60. PerfEvtSel0 and PerfEvtSel1 MSRs

- **USR (user mode) flag (bit 16)** — Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19)** — When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20)** — When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31)** — When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

18.23.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

18.23.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

18.23.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking
- Initialize and start counters
- Stop counters
- Read the event counters
- Read the time-stamp counter

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 18.23.3, “Starting and Stopping the Performance-Monitoring Counters”).

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

18.23.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

18.24 PERFORMANCE MONITORING (PENTIUM PROCESSORS)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed in Table 19-39 are model-specific for the Pentium processor.

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

18.24.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 18-61). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21)** — Selects (by entering an event code in the field) up to two events to be monitored. See Table 19-39 for a list of available event codes.

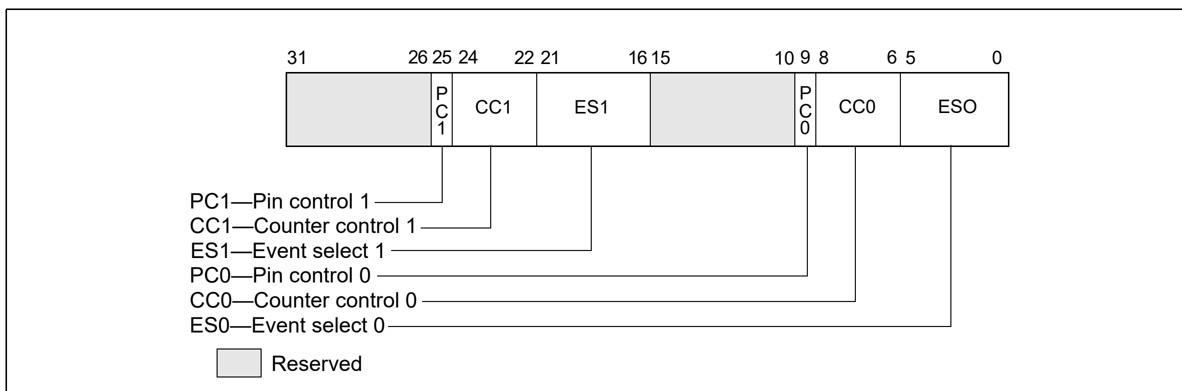


Figure 18-61. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24)** — Controls the operation of the counter. Control codes are as follows:

  000 — Count nothing (counter disabled)

  001 — Count the selected event while CPL is 0, 1, or 2

  010 — Count the selected event while CPL is 3

  011 — Count the selected event regardless of CPL

  100 — Count nothing (counter disabled)

  101 — Count clocks (duration) while CPL is 0, 1, or 2

  110 — Count clocks (duration) while CPL is 3

  111 — Count clocks (duration) regardless of CPL

  The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25)** — Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the

overflow or incremented condition. The external signalling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

18.24.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an "occurrence event" is being counted, the associated pin is asserted (high) each time the event occurs. When a "duration event" is being counted, the associated PM pin is asserted for the entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A "count up to" function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less then $2^{40} - 1$. After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

18.24.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

13. Updates to Chapter 19, Volume 3B

Change bars show changes to Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Added performance monitoring event 3EH to the Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture.

CHAPTER 19
PERFORMANCE-MONITORING EVENTS

This chapter lists the performance-monitoring events that can be monitored with the Intel 64 or IA-32 processors. The ability to monitor performance events and the events that can be monitored in these processors are mostly model-specific, except for architectural performance events, described in Section 19.1.

Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture:

- Section 19.2 - Processors based on Skylake and Kaby Lake microarchitectures
- Section 19.3 - Processors based on Knights Landing microarchitecture
- Section 19.4 - Processors based on Broadwell microarchitecture
- Section 19.5 - Processors based on Haswell microarchitecture
- Section 19.5.1 - Processors based on Haswell-E microarchitecture
- Section 19.6 - Processors based on Ivy Bridge microarchitecture
- Section 19.6.1 - Processors based on Ivy Bridge-E microarchitecture
- Section 19.7 - Processors based on Sandy Bridge microarchitecture
- Section 19.8 - Processors based on Intel® microarchitecture code name Nehalem
- Section 19.9 - Processors based on Intel® microarchitecture code name Westmere
- Section 19.10 - Processors based on Enhanced Intel® Core™ microarchitecture
- Section 19.11 - Processors based on Intel® Core™ microarchitecture
- Section 19.12 - Processors based on the Goldmont microarchitecture
- Section 19.13 - Processors based on the Silvermont microarchitecture
- Section 19.13.1 - Processors based on the Airmont microarchitecture
- Section 19.14 - 45 nm and 32 nm Intel® Atom™ Processors
- Section 19.15 - Intel® Core™ Solo and Intel® Core™ Duo processors
- Section 19.16 - Processors based on Intel NetBurst® microarchitecture
- Section 19.17 - Pentium® M family processors
- Section 19.18 - P6 family processors
- Section 19.19 - Pentium® processors

NOTE

These performance-monitoring events are intended to be used as guides for performance tuning. The counter values reported by the performance-monitoring events are approximate and believed to be useful as relative guides for tuning software. Known discrepancies are documented where applicable.

All performance event encodings not documented in the appropriate tables for the given processor are considered reserved, and their use will result in undefined counter updates with associated overflow actions.

The event tables listed this chapter provide information for tool developers to support architectural and non-architectural performance monitoring events. The tables are up to date at processor launch, but are subject to changes. The most up to date event tables and additional details of performance event implementation for end-user (including additional details beyond event code/umask) can found at the “perfmon” repository provided by The Intel Open Source Technology Center (https://download.01.org/perfmon/).

19.1 ARCHITECTURAL PERFORMANCE-MONITORING EVENTS

Architectural performance events are introduced in Intel Core Solo and Intel Core Duo processors. They are also supported on processors based on Intel Core microarchitecture. Table 19-1 lists pre-defined architectural performance events that can be configured using general-purpose performance counters and associated event-select registers.

Table 19-1. Architectural Performance Events

Event Num.	Event Mask Name	Umask Value	Description
3CH	UnHalted Core Cycles	00H	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
3CH	UnHalted Reference Cycles[1]	01H	Counts at a fixed frequency whenever the logical processor is in C0 state (not halted).
C0H	Instructions Retired	00H	Counts when the last uop of an instruction retires.
2EH	LLC Reference	4FH	Counts requests originating from the core that reference a cache line in the last level on-die cache.
2EH	LLC Misses	41H	Counts each cache miss condition for references to the last level on-die cache.
C4H	Branch Instruction Retired	00H	Counts when the last uop of a branch instruction retires.
C5H	Branch Misses Retired	00H	Counts when the last uop of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.

NOTES:

1. Current implementations count at core crystal clock, TSC, or bus clock frequency.

Fixed-function performance counters count only events defined in Table 19-2.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD/CPU_CLK_UNHALTED.CORE/CPU_CLK_UNHALTED.THREAD_ANY	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. If there are more than one logical processor in a processor core, CPU_CLK_UNHALTED.THREAD_ANY is supported by programming IA32_FIXED_CTR_CTRL[bit 6]AnyThread = 1. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events (Contd.)

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

19.2 PERFORMANCE MONITORING EVENTS FOR 6TH GENERATION INTEL® CORE™ PROCESSOR AND 7TH GENERATION INTEL® CORE™ PROCESSOR

6th Generation Intel® Core™ processors are based on the Skylake microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Fixed counters in the core PMU support the architecture events defined in Table 19-2. Non-architectural performance-monitoring events in the processor core are listed in Table 19-3. The events in Table 19-3 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_4EH and 06_5EH. Table 19-9 lists performance events supporting Intel TSX (see Section 18.11.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-9, they are listed in Table 19-4.

7th Generation Intel® Core™ processors are based on the Kaby Lake microarchitecture. Non-architectural performance-monitoring events in the processor core are listed in Table 19-3. The events in Table 19-3 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_8EH and 06_9EH.

The comment column in Table 19-3 uses abbreviated letters to indicate additional conditions applicable to the Event Mask Mnemonic. For event umasks listed in Table 19-3 that do not show “AnyT”, users should refrain from programming “AnyThread =1” in IA32_PERF_EVTSELx.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Load misses in all TLB levels causes a page walk that completes. (All page sizes.)	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a load.	
08H	10H	DTLB_LOAD_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a walk for a load.	CMSK1
08H	20H	DTLB_LOAD_MISSES.STLB_HIT	Loads that miss the DTLB but hit STLB.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0DH	01H	INT_MISC.RECOVERY_CYCLES	Core cycles the allocator was stalled due to recovery from earlier machine clear event for this thread (for example, misprediction or memory order conflict).	
0DH	01H	INT_MISC.RECOVERY_CYCLES_ANY	Core cycles the allocator was stalled due to recovery from earlier machine clear event for any logical thread in this processor core.	AnyT
0DH	80H	INT_MISC.CLEAR_RESTEER_CYCLES	Cycles the issue-stage is waiting for front end to fetch from resteered path following branch misprediction or machine clear events.	
0EH	01H	UOPS_ISSUED.ANY	The number of uops issued by the RAT to RS.	
0EH	01H	UOPS_ISSUED.STALL_CYCLES	Cycles when the RAT does not issue uops to RS for the thread.	CMSK1, INV
0EH	02H	UOPS_ISSUED.VECTOR_WIDTH_MISMATCH	Uops inserted at issue-stage in order to preserve upper bits of vector registers.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
14H	01H	ARITH.FPU_DIVIDER_ACTIVE	Cycles when divider is busy executing divide or square root operations. Accounts for FP operations including integer divides.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand Data Read requests that missed L2, no rejects.	
24H	22H	L2_RQSTS.RFO_MISS	RFO requests that missed L2.	
24H	24H	L2_RQSTS.CODE_RD_MISS	L2 cache misses when fetching instructions.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that missed L2.	
24H	38H	L2_RQSTS.PF_MISS	Requests from the L1/L2/L3 hardware prefetchers or load software prefetches that miss L2 cache.	
24H	3FH	L2_RQSTS.MISS	All requests that missed L2.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	42H	L2_RQSTS.RFO_HIT	RFO requests that hit L2 cache.	
24H	44H	L2_RQSTS.CODE_RD_HIT	L2 cache hits when fetching instructions.	
24H	D8H	L2_RQSTS.PF_HIT	Prefetches that hit L2.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	All demand data read requests to L2.	
24H	E2H	L2_RQSTS.ALL_RFO	All L RFO requests to L2.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	All L2 code requests.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	All demand requests to L2.	
24H	F8H	L2_RQSTS.ALL_PF	All requests from the L1/L2/L3 hardware prefetchers or load software prefetches.	
24H	EFH	L2_RQSTS.REFERENCES	All requests to L2.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the L3 cache.	See Table 19-1.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the L3 cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Cycles while the logical processor is not in a halt state.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P_ANY	Cycles while at least one logical processor is not in a halt state.	AnyT
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyT
3CH	02H	CPU_CLK_THREAD_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle.	
48H	01H	L1D_PEND_MISS.PENDING_CYCLES	Cycles with at least one outstanding L1D misses from this logical processor.	CMSK1
48H	01H	L1D_PEND_MISS.PENDING_CYCLES_ANY	Cycles with at least one outstanding L1D misses from any logical processor in this core.	CMSK1, AnyT
48H	02H	L1D_PEND_MISS.FB_FULL	Number of times a request needed a FB entry but there was no entry available for it. That is, the FB unavailability was the dominant reason for blocking the request. A request includes cacheable/uncacheable demand that is load, store or SW prefetch. HWP are excluded.	
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Store misses in all TLB levels that cause page walks.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Counts completed page walks in any TLB levels due to store misses (all page sizes).	
49H	10H	DTLB_STORE_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a store.	
49H	10H	DTLB_STORE_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a page walk for a store.	CMSK1
49H	20H	DTLB_STORE_MISSES.STLB_HIT	Store misses that missed DTLB but hit STLB.	
4CH	01H	LOAD_HIT_PRE.HW_PF	Demand load dispatches that hit fill buffer allocated for software prefetch.	
4FH	10H	EPT.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with an EPT walk for any request type.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
5EH	01H	RS_EVENTS.EMPTY_END	Counts end of periods where the Reservation Station (RS) was empty. Could be useful to precisely locate Front-end Latency Bound issues.	CMSK1, INV
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Increment each cycle of the number of offcore outstanding Demand Data Read transactions in SQ to uncore.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_DATA_RD	Cycles with at least one offcore outstanding Demand Data Read transactions in SQ to uncore.	CMSK1
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD_GE_6	Cycles with at least 6 offcore outstanding Demand Data Read transactions in SQ to uncore.	CMSK6
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Increment each cycle of the number of offcore outstanding demand code read transactions in SQ to uncore.	
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_CODE_RD	Cycles with at least one offcore outstanding demand code read transactions in SQ to uncore.	CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Increment each cycle of the number of offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO	Cycles with at least one offcore outstanding RFO transactions in SQ to uncore.	CMSK1
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Increment each cycle of the number of offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DATA_RD	Cycles with at least one offcore outstanding data read transactions in SQ to uncore.	CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD	Increment each cycle of the number of offcore outstanding demand data read requests from SQ that missed L3.	
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_L3_MISS_DEMAND_DATA_RD	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD_GE_6	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK6
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path.	
79H	04H	IDQ.MITE_CYCLES	Cycles when uops are being delivered to IDQ from MITE path.	CMSK1
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path.	
79H	08H	IDQ.DSB_CYCLES	Cycles when uops are being delivered to IDQ from DSB path.	CMSK1
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ by DSB when MS_busy.	
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Cycles DSB is delivered at least one uops.	CMSK1
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Cycles DSB is delivered four uops.	CMSK4

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ by MITE when MS_busy.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops.	CMSK1
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops.	CMSK4
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ while MS is busy.	
79H	30H	IDQ.MS_SWITCHES	Number of switches from DSB or MITE to MS.	EDG
79H	30H	IDQ.MS_CYCLES	Cycles MS is delivered at least one uops.	CMSK1
80H	04H	ICACHE_16B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache miss.	
80H	04H	ICACHE_64B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	
83H	01H	ICACHE_64B.IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	02H	ICACHE_64B.IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses at all ITLB levels that cause page walks.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Counts completed page walks in any TLB level due to code fetch misses (all page sizes).	
85H	10H	ITLB_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for an instruction fetch request.	
85H	20H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOP_DELIV.CORE	Cycles which 4 issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	CMSK4
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_n_UOP_DELIV.CORE	Cycles which “4-n” issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	Set CMSK = 4-n; n = 1, 2, 3
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	Cycles which front end delivered 4 uops or the RAT was stalling FE.	CMSK, INV
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	
A2H	01H	RESOURCE_STALLS.ANY	Resource-related stall cycles.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_MISS	Cycles while L2 cache miss demand load is outstanding.	CMSK1
A3H	02H	CYCLE_ACTIVITY.CYCLES_L3_MISS	Cycles while L3 cache miss demand load is outstanding.	CMSK2
A3H	04H	CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls.	CMSK4
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_MISS	Execution stalls while L2 cache miss demand load is outstanding.	CMSK5
A3H	06H	CYCLE_ACTIVITY.STALLS_L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	CMSK6
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_MISS	Cycles while L1 data cache miss demand load is outstanding.	CMSK8
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_MISS	Execution stalls while L1 data cache miss demand load is outstanding.	CMSK12
A3H	10H	CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load.	CMSK16
A3H	14H	CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load.	CMSK20
A6H	01H	EXE_ACTIVITY.EXE_BOUND_0_PORTS	Cycles for which no uops began execution, the Reservation Station was not empty, the Store Buffer was full and there was no outstanding load.	
A6H	02H	EXE_ACTIVITY.1_PORTS_UTIL	Cycles for which one uop began execution on any port, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.2_PORTS_UTIL	Cycles for which two uops began execution, and the Reservation Station was not empty.	
A6H	08H	EXE_ACTIVITY.3_PORTS_UTIL	Cycles for which three uops began execution, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.4_PORTS_UTIL	Cycles for which four uops began execution, and the Reservation Station was not empty.	
A6H	40H	EXE_ACTIVITY.BOUND_ON_STORES	Cycles where the Store Buffer was full and no outstanding load.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
A8H	01H	LSD.CYCLES_ACTIVE	Cycles with at least one uop delivered by the LSD and none from the decoder.	CMSK1

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A8H	01H	LSD.CYCLES_4_UOPS	Cycles with 4 uops delivered by the LSD and none from the decoder.	CMSK4
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	DSB-to-MITE switch true penalty cycles.	
AEH	01H	ITLB.ITLB_FLUSH	Flushing of the Instruction TLB (ITLB) pages, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B0H	10H	OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD	Demand data read requests that missed L3.	
B0H	80H	OFFCORE_REQUESTS.ALL_REQUESTS	Any memory transaction that reached the SQ.	
B1H	01H	UOPS_EXECUTED.THREAD	Counts the number of uops that begin execution across all ports.	
B1H	01H	UOPS_EXECUTED.STALL_CYCLES	Cycles where there were no uops that began execution.	CMSK, INV
B1H	01H	UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC	Cycles where there was at least one uop that began execution.	CMSK1
B1H	01H	UOPS_EXECUTED.CYCLES_GE_2_UOP_EXEC	Cycles where there were at least two uops that began execution.	CMSK2
B1H	01H	UOPS_EXECUTED.CYCLES_GE_3_UOP_EXEC	Cycles where there were at least three uops that began execution.	CMSK3
B1H	01H	UOPS_EXECUTED.CYCLES_GE_4_UOP_EXEC	Cycles where there were at least four uops that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE	Counts the number of uops from any logical processor in this core that begin execution.	
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_1	Cycles where there was at least one uop, from any logical processor in this core, that began execution.	CMSK1
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_2	Cycles where there were at least two uops, from any logical processor in this core, that began execution.	CMSK2
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_3	Cycles where there were at least three uops, from any logical processor in this core, that began execution.	CMSK3
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_4	Cycles where there were at least four uops, from any logical processor in this core, that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_NONE	Cycles where there were no uops from any logical processor in this core that began execution.	CMSK1, INV
B1H	10H	UOPS_EXECUTED.X87	Counts the number of X87 uops that begin execution.	
B2H	01H	OFF_CORE_REQUEST_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this core.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.9.5, “Off-core Response Performance Monitoring”.	Requires MSR 01A6H
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.9.5, “Off-core Response Performance Monitoring”.	Requires MSR 01A7H
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	01H	TLB_FLUSH.STLB_ANY	STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only;
C0H	01H	INST_RETIRED.TOTAL_CYCLES	Number of cycles using always true condition applied to PEBS instructions retired event.	CMSK10, PS
C1H	3FH	OTHER_ASSISTS.ANY	Number of times a microcode assist is invoked by HW other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists.	
C2H	01H	UOPS_RETIRED.STALL_CYCLES	Cycles without actually retired uops.	CMSK1, INV
C2H	01H	UOPS_RETIRED.TOTAL_CYCLES	Cycles with less than 10 actually retired uops.	CMSK10, INV
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Retirement slots used.	
C3H	01H	MACHINE_CLEARS.COUNT	Number of machine clears of any type.	CMSK1, EDG
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions that retired.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	PS
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	PS
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	PS
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	PS
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	PS
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	PS
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were mispredicted and taken.	PS

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C6H	01H	FRONTEND_RETIRED.DSB_MISS	Retired instructions which experienced DSB miss. Specify MSR_PEBS_FRONTEND.EVTSEL=11H.	PS
C6H	01H	FRONTEND_RETIRED.L1I_MISS	Retired instructions which experienced instruction L1 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=12H.	PS
C6H	01H	FRONTEND_RETIRED.L2_MISS	Retired instructions which experienced L2 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=13H.	PS
C6H	01H	FRONTEND_RETIRED.ITLB_MISS	Retired instructions which experienced ITLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=14H.	PS
C6H	01H	FRONTEND_RETIRED.STLB_MISS	Retired instructions which experienced STLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=15H.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_16	Retired instructions that are fetched after an interval where the front end delivered no uops for at least 16 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =16, IDQ_Bubble_Width = 4.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_m	Retired instructions that are fetched after an interval where the front end had 'm' IDQ slots delivered, no uops for at least 2 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =2, IDQ_Bubble_Width = m.	PS, m = 1, 2, 3
C7H	01H	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of double-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one DP FLOP.
C7H	02H	FP_ARITH_INST_RETIRED.SCALAR_SINGLE	Number of single-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one SP FLOP.
C7H	04H	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of double-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPD instruction counts as 2.	Software may treat each count as two DP FLOPs.
C7H	08H	FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	Number of single-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPS instruction counts as 2.	Software may treat each count as four SP FLOPs.
C7H	10H	FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of double-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA instruction counts as 2.	Software may treat each count as four DP FLOPs.
C7H	20H	FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	Number of single-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA or VDPPS instruction counts as 2.	Software may treat each count as eight SP FLOPs.
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	CMSK1
CBH	01H	HW_INTERRUPTS.RECEIVED	Number of hardware interrupts received by the processor.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PSDLA
D0H	11H	MEM_INST_RETIRED.STLB_MISS_LOADS	Retired load instructions that miss the STLB.	PSDLA

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D0H	12H	MEM_INST_RETIRED.STLB_MISS_STORES	Retired store instructions that miss the STLB.	PSDLA
D0H	21H	MEM_INST_RETIRED.LOCK_LOADS	Retired load instructions with locked access.	PSDLA
D0H	41H	MEM_INST_RETIRED.SPLIT_LOADS	Number of load instructions retired with cache-line splits that may impact performance.	PSDLA
D0H	42H	MEM_INST_RETIRED.SPLIT_STORES	Number of store instructions retired with line-split.	PSDLA
D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	PSDLA
D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	PSDLA
D1H	01H	MEM_LOAD_RETIRED.L1_HIT	Retired load instructions with L1 cache hits as data sources.	PSDLA
D1H	02H	MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources.	PSDLA
D1H	04H	MEM_LOAD_RETIRED.L3_HIT	Retired load instructions with L3 cache hits as data sources.	PSDLA
D1H	08H	MEM_LOAD_RETIRED.L1_MISS	Retired load instructions missed L1 cache as data sources.	PSDLA
D1H	10H	MEM_LOAD_RETIRED.L2_MISS	Retired load instructions missed L2. Unknown data source excluded.	PSDLA
D1H	20H	MEM_LOAD_RETIRED.L3_MISS	Retired load instructions missed L3. Excludes unknown data source.	PSDLA
D1H	40H	MEM_LOAD_RETIRED.FB_HIT	Retired load instructions where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	PSDLA
D2H	01H	MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS	Retired load instructions where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	PSDLA
D2H	02H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT	Retired load Instructions where data sources were L3 and cross-core snoop hits in on-pkg core cache.	PSDLA
D2H	04H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM	Retired load instructions where data sources were HitM responses from shared L3.	PSDLA
D2H	08H	MEM_LOAD_L3_HIT_RETIRED.XSNP_NONE	Retired load instructions where data sources were hits in L3 without snoops required.	PSDLA
E6H	01H	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	
CMSK1: Counter Mask = 1 required; CMSK4: CounterMask = 4 required; CMSK6: CounterMask = 6 required; CMSK8: CounterMask = 8 required; CMSK10: CounterMask = 10 required; CMSK12: CounterMask = 12 required; CMSK16: CounterMask = 16 required; CMSK20: CounterMask = 20 required. AnyT: AnyThread = 1 required. INV: Invert = 1 required. EDG: EDGE = 1 required. PSDLA: Also supports PEBS and DataLA. PS: Also supports PEBS.				

Table 19-9 lists performance events supporting Intel TSX (see Section 18.11.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-9, they are listed in Table 19-4.

Table 19-4. Intel® TSX Performance Event Addendum in Processors based on Skylake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.3 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON PHI™ PROCESSOR 3200, 5200, 7200 SERIES

Intel® Xeon Phi™ processors 3200/5200/7200 series are based on the Knights Landing microarchitecture. Non-architectural performance-monitoring events in the processor core are listed in Table 19-5. The events in Table 19-5 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following value 06_57H.

Table 19-5. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	RECYCLEQ.LD_BLOCK_ST_FORWARD	Counts the number of occurrences a retired load gets blocked because its address partially overlaps with a store.	PSDLA
03H	02H	RECYCLEQ.LD_BLOCK_STD_NOTREADY	Counts the number of occurrences a retired load gets blocked because its address overlaps with a store whose data is not ready.	
03H	04H	RECYCLEQ.ST_SPLITS	Counts the number of occurrences a retired store that is a cache line split. Each split should be counted only once.	
03H	08H	RECYCLEQ.LD_SPLITS	Counts the number of occurrences a retired load that is a cache line split. Each split should be counted only once.	PSDLA

Table 19-5. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	10H	RECYCLEQ.LOCK	Counts all the retired locked loads. It does not include stores because we would double count if we count stores.	
03H	20H	RECYCLEQ.STA_FULL	Counts the store micro-ops retired that were pushed in the recycle queue because the store address buffer is full.	
03H	40H	RECYCLEQ.ANY_LD	Counts any retired load that was pushed into the recycle queue for any reason.	
03H	80H	RECYCLEQ.ANY_ST	Counts any retired store that was pushed into the recycle queue for any reason.	
04H	01H	MEM_UOPS_RETIRED.L1_MISS_LOADS	Counts the number of load micro-ops retired that miss in L1 D cache.	
04H	02H	MEM_UOPS_RETIRED.L2_HIT_LOADS	Counts the number of load micro-ops retired that hit in the L2.	PSDLA
04H	04H	MEM_UOPS_RETIRED.L2_MISS_LOADS	Counts the number of load micro-ops retired that miss in the L2.	PSDLA
04H	08H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Counts the number of load micro-ops retired that cause a DTLB miss.	PSDLA
04H	10H	MEM_UOPS_RETIRED.UTLB_MISS_LOADS	Counts the number of load micro-ops retired that caused micro TLB miss.	
04H	20H	MEM_UOPS_RETIRED.HITM	Counts the loads retired that get the data from the other core in the same tile in M state.	
04H	40H	MEM_UOPS_RETIRED.ALL_LOADS	Counts all the load micro-ops retired.	
04H	80H	MEM_UOPS_RETIRED.ALL_STORES	Counts all the store micro-ops retired.	
05H	01H	PAGE_WALKS.D_SIDE_WALKS	Counts the total D-side page walks that are completed or started. The page walks started in the speculative path will also be counted.	EdgeDetect=1
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts the total number of core cycles for all the D-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	02H	PAGE_WALKS.I_SIDE_WALKS	Counts the total I-side page walks that are completed.	EdgeDetect=1
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts the total number of core cycles for all the I-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	03H	PAGE_WALKS.WALKS	Counts the total page walks that are completed (I-side and D-side).	EdgeDetect=1
05H	03H	PAGE_WALKS.CYCLES	Counts the total number of core cycles for all the page walks. The cycles for page walks started in speculative path will also be included.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts the number of L2 cache misses. Also called L2_REQUESTS_MISS.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts the total number of L2 cache references. Also called L2_REQUESTS_REFERENCE.	

Table 19-5. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
30H	00H	L2_REQUESTS_REJECT.ALL	Counts the number of MEC requests from the L2Q that reference a cache line (cacheable requests) excluding SW prefetches filling only to L2 cache and L1 evictions (automatically excludes L2HWP, UC, WC) that were rejected - Multiple repeated rejects should be counted multiple times.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of MEC requests that were not accepted into the L2Q because of any L2 queue reject condition. There is no concept of at-ret here. It might include requests due to instructions in the speculative path.	
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of unhalted core clock cycles.	
3CH	01H	CPU_CLK_UNHALTED.REF	Counts the number of unhalted reference clock cycles.	
3EH	04H	L2_PREFETCHER.ALLOC_XQ	Counts the number of L2HWP allocated into XQ GP.	
80H	01H	ICACHE.HIT	Counts all instruction fetches that hit the instruction cache.	
80H	02H	ICACHE.MISSES	Counts all instruction fetches that miss the instruction cache or produce memory requests. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	ICACHE.ACCESSES	Counts all instruction fetches, including uncacheable fetches.	
86H	04H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts the number of core cycles the fetch stalls because of an icache miss. This is a cumulative count of core cycles the fetch stalled for all icache misses.	
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.14.1.2.	Requires MSR_OFFCORE_RESP 0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.14.1.2.	Requires MSR_OFFCORE_RESP 1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Counts the total number of instructions retired.	PS
C2H	01H	UOPS_RETIRED.MS	Counts the number of micro-ops retired that are from the complex flows issued by the micro-sequencer (MS).	
C2H	10H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired.	
C2H	20H	UOPS_RETIRED.SCALAR_SIMD	Counts the number of scalar SSE, AVX, AVX2, and AVX-512 micro-ops except for loads (memory-to-register mov-type micro ops), division and sqrt.	
C2H	40H	UOPS_RETIRED.PACKED_SIMD	Counts the number of packed SSE, AVX, AVX2, and AVX-512 micro-ops (both floating point and integer) except for loads (memory-to-register mov-type micro-ops), packed byte and word multiplies.	
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the machine clears due to program modifying data within 1K of a recently fetched code page.	

Table 19-5. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of times the machine clears due to memory ordering hazards.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts the number of floating operations retired that required microcode assists.	
C3H	08H	MACHINE_CLEARS.ALL	Counts all machine clears.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	7EH	BR_INST_RETIRED.JCC	Counts the number of JCC branch instructions retired.	PS
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts the number of far branch instructions retired.	PS
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts the number of branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C4H	F7H	BR_INST_RETIRED.RETURN	Counts the number of near RET branch instructions retired.	PS
C4H	F9H	BR_INST_RETIRED.CALL	Counts the number of near CALL branch instructions retired.	PS
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts the number of near indirect CALL branch instructions retired.	PS
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts the number of near relative CALL branch instructions retired.	PS
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts the number of branch instructions retired that were taken conditional jumps.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts the number of mispredicted branch instructions retired.	PS
C5H	7EH	BR_MISP_RETIRED.JCC	Counts the number of mispredicted JCC branch instructions retired.	PS
C5H	BFH	BR_MISP_RETIRED.FAR_BRANCH	Counts the number of mispredicted far branch instructions retired.	PS
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts the number of mispredicted branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts the number of mispredicted near RET branch instructions retired.	PS
C5H	F9H	BR_MISP_RETIRED.CALL	Counts the number of mispredicted near CALL branch instructions retired.	PS
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts the number of mispredicted near indirect CALL branch instructions retired.	PS
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Counts the number of mispredicted near relative CALL branch instructions retired.	PS
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts the number of mispredicted branch instructions retired that were taken conditional jumps.	PS
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of core cycles when no micro-ops are allocated and the ROB is full.	
CAH	04H	NO_ALLOC_CYCLES.MISPREDICTS	Counts the number of core cycles when no micro-ops are allocated and the alloc pipe is stalled waiting for a mispredicted branch to retire.	

Table 19-5. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of core cycles when no micro-ops are allocated and a RATstall (caused by reservation station full) is asserted.	
CAH	90H	NO_ALLOC_CYCLES.NOT_DELIVERED	Counts the number of core cycles when no micro-ops are allocated, the IQ is empty, and no other condition is blocking allocation.	
CAH	7FH	NO_ALLOC_CYCLES.ALL	Counts the total number of core cycles when no micro-ops are allocated for any reason.	
CBH	01H	RS_FULL_STALL.MEC	Counts the number of core cycles when allocation pipeline is stalled and is waiting for a free MEC reservation station entry.	
CBH	1FH	RS_FULL_STALL.ALL	Counts the total number of core cycles the allocation pipeline is stalled when any one of the reservation stations is full.	
CDH	01H	CYCLES_DIV_BUSY.ALL	Cycles the number of core cycles when divider is busy. Does not imply a stall waiting for the divider.	
E6H	01H	BACLEARS.ALL	Counts the number of times the front end resteers for any branch as a result of another branch handling mechanism in the front end.	
E6H	08H	BACLEARS.RETURN	Counts the number of times the front end resteers for RET branches as a result of another branch handling mechanism in the front end.	
E6H	10H	BACLEARS.COND	Counts the number of times the front end resteers for conditional branches as a result of another branch handling mechanism in the front end.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the MSROM starts a flow of uops.	
PS: Also supports PEBS. PSDLA: Also supports PEBS and DataLA.				

19.4 PERFORMANCE MONITORING EVENTS FOR THE INTEL® CORE™ M AND 5TH GENERATION INTEL® CORE™ PROCESSORS

The Intel® Core™ M processors, the 5th generation Intel® Core™ processors and the Intel Xeon processor E3 1200 v4 product family are based on the Broadwell microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-6. The events in Table 19-6 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3DH and 06_47H. Table 19-9 lists performance events supporting Intel TSX (see Section 18.11.5) and the events are available on processors based on Broadwell microarchitecture. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Broadwell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3DH and 06_47H support uncore performance events listed in Table 19-10.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SiNGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles when divider is busy executing divide operations.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4FH	10H	EPT.WALK_CYCLES	Cycles of Extended Page Table walks.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding demand code read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	Number of uops delivered to IDQ from any path.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that cause a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	Set AnyThread to count per core.
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	Set AnyThread to count per core.
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	Set AnyThread to count per core.
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	Set AnyThread to count per core.
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	Set AnyThread to count per core.
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	Set AnyThread to count per core.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	Set AnyThread to count per core.
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles of delay due to Decode Stream Buffer to MITE switches.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-logical-processor each cycle.	Use Cmask to count stall cycles.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	02H	INST_RETIRED.X87	FP operations retired. X87 FP operations that have no exceptions.	
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	01H	MACHINE_CLEARS.CYCLES	Counts cycles while a machine clears stalled forward progress of a logical processor or a processor core.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source.	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops where data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops where data sources were HitM responses from shared L3.	Supports PEBS and DataLA.
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops where data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops where data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	

Table 19-9 lists performance events supporting Intel TSX (see Section 18.11.5) and the events are applicable to processors based on Broadwell microarchitecture. Where Broadwell microarchitecture implements TSX-related event semantics that differ from Table 19-9, they are listed in Table 19-7.

Table 19-7. Intel® TSX Performance Event Addendum in Processors Based on Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.5 PERFORMANCE MONITORING EVENTS FOR THE 4TH GENERATION INTEL® CORE™ PROCESSORS

4th generation Intel® Core™ processors and Intel Xeon processor E3-1200 v3 product family are based on the Haswell microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-8. The events in Table 19-8 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3CH, 06_45H and 06_46H. Table 19-9 lists performance events focused on supporting Intel TSX (see Section 18.11.5). Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g., derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to demand load misses that caused 2M/4M page walks in any TLB levels.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Completed page walks in any TLB of any page size due to demand load misses.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
08H	40H	DTLB_LOAD_MISSES.STLB_HIT_2M	Load misses that missed DTLB but hit STLB (2M).	
08H	60H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
08H	80H	DTLB_LOAD_MISSES.PDE_CACHE_MISS	DTLB demand load misses with low part of linear-to-physical address translation missed.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1 to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SiNGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	42H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache.	
24H	22H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	44H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	24H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that miss L2 cache.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	Demand requests to L2 cache.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
24H	3FH	L2_RQSTS.MISS	All requests that missed L2.	
24H	FFH	L2_RQSTS.REFERENCES	All requests to L2 cache.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to store misses in one or more TLB levels of 2M/4M page structure.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Completed page walks due to store miss in any TLB levels of any page size (4K/2M/4M/1G).	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
49H	40H	DTLB_STORE_MISSES.STLB_HIT_2M	Store misses that missed DTLB but hit STLB (2M).	
49H	60H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
49H	80H	DTLB_STORE_MISSES.PDE_CACHE_MISS	DTLB store misses with low part of linear-to-physical address translation missed.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that causes a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to misses in ITLB 2M/4M page entries.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Completed page walks in ITLB of any page size.	
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
85H	40H	ITLB_MISSES.STLB_HIT_2M	ITLB misses that hit STLB (2M).	
85H	60H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_EXECUTED_PORT.PORT_0	Cycles which a uop is dispatched on port 0 in this thread.	Set AnyThread to count per core.
A1H	02H	UOPS_EXECUTED_PORT.PORT_1	Cycles which a uop is dispatched on port 1 in this thread.	Set AnyThread to count per core.
A1H	04H	UOPS_EXECUTED_PORT.PORT_2	Cycles which a uop is dispatched on port 2 in this thread.	Set AnyThread to count per core.
A1H	08H	UOPS_EXECUTED_PORT.PORT_3	Cycles which a uop is dispatched on port 3 in this thread.	Set AnyThread to count per core.
A1H	10H	UOPS_EXECUTED_PORT.PORT_4	Cycles which a uop is dispatched on port 4 in this thread.	Set AnyThread to count per core.
A1H	20H	UOPS_EXECUTED_PORT.PORT_5	Cycles which a uop is dispatched on port 5 in this thread.	Set AnyThread to count per core.
A1H	40H	UOPS_EXECUTED_PORT.PORT_6	Cycles which a uop is dispatched on port 6 in this thread.	Set AnyThread to count per core.
A1H	80H	UOPS_EXECUTED_PORT.PORT_7	Cycles which a uop is dispatched on port 7 in this thread	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set Cmask=2 to count cycle.	Use only when HTT is off.
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set Cmask=2 to count cycle.	
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Use only when HTT is off.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 data cache miss loads. Set Cmask=8 to count cycle.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFF_CORE_RESPONSE_0	See Table 18-48 or Table 18-49.	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Table 18-48 or Table 18-49.	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
BCH	28H	PAGE_WALKER_LOADS.ITLB_MEMORY	Number of ITLB page walker loads from memory.	
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use Cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA; use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were taken but mispredicted.	
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source .	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared L3.	Supports PEBS and DataLA.
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.

Table 19-8. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	06H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	

Table 19-9. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	01H	TX_MEM.ABORT_CONFLICT	Number of times a transactional abort was signaled due to a data conflict on a transactionally accessed address.	
54H	02H	TX_MEM.ABORT_CAPACITY_WRITE	Number of times a transactional abort was signaled due to a data capacity limitation for transactional writes.	
54H	04H	TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a HLE transactional region aborted due to a non XRELEASE prefixed instruction writing to an elided lock in the elision buffer.	
54H	08H	TX_MEM.ABORT_HLE_ELISION_BUFFER_NOT_EMPTY	Number of times an HLE transactional execution aborted due to NoAllocatedElisionBuffer being non-zero.	
54H	10H	TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH	Number of times an HLE transactional execution aborted due to XRELEASE lock not satisfying the address and value requirements in the elision buffer.	
54H	20H	TX_MEM.ABORT_HLE_ELISION_BUFFER_UNSUPPORTED_ALIGNMENT	Number of times an HLE transactional execution aborted due to an unsupported read alignment from the elision buffer.	
54H	40H	TX_MEM.HLE_ELISION_BUFFER_FULL	Number of times HLE lock could not be elided due to ElisionBufferAvailable being zero.	

Table 19-9. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
5DH	01H	TX_EXEC.MISC1	Counts the number of times a class of instructions that may cause a transactional abort was executed. Since this is the count of execution, it may not always cause a transactional abort.	
5DH	02H	TX_EXEC.MISC2	Counts the number of times a class of instructions (for example, vzeroupper) that may cause a transactional abort was executed inside a transactional region.	
5DH	04H	TX_EXEC.MISC3	Counts the number of times an instruction execution caused the transactional nest count supported to be exceeded.	
5DH	08H	TX_EXEC.MISC4	Counts the number of times an XBEGIN instruction was executed inside an HLE transactional region.	
5DH	10H	TX_EXEC.MISC5	Counts the number of times an instruction with HLE-XACQUIRE semantic was executed inside an RTM transactional region.	
C8H	01H	HLE_RETIRED.START	Number of times an HLE execution started.	IF HLE is supported.
C8H	02H	HLE_RETIRED.COMMIT	Number of times an HLE execution successfully committed.	
C8H	04H	HLE_RETIRED.ABORTED	Number of times an HLE execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	
C8H	08H	HLE_RETIRED.ABORTED_MEM	Number of times an HLE execution aborted due to various memory events (for example, read/write capacity and conflicts).	
C8H	10H	HLE_RETIRED.ABORTED_TIMER	Number of times an HLE execution aborted due to uncommon conditions.	
C8H	20H	HLE_RETIRED.ABORTED_UNFRIENDLY	Number of times an HLE execution aborted due to HLE-unfriendly instructions.	
C8H	40H	HLE_RETIRED.ABORTED_MEMTYPE	Number of times an HLE execution aborted due to incompatible memory type.	
C8H	80H	HLE_RETIRED.ABORTED_EVENTS	Number of times an HLE execution aborted due to none of the previous 4 categories (for example, interrupts).	
C9H	01H	RTM_RETIRED.START	Number of times an RTM execution started.	IF RTM is supported.
C9H	02H	RTM_RETIRED.COMMIT	Number of times an RTM execution successfully committed.	
C9H	04H	RTM_RETIRED.ABORTED	Number of times an RTM execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	

Table 19-9. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C9H	08H	RTM_RETIRED.ABORTED_MEM	Number of times an RTM execution aborted due to various memory events (for example, read/write capacity and conflicts).	IF RTM is supported.
C9H	10H	RTM_RETIRED.ABORTED_TIMER	Number of times an RTM execution aborted due to uncommon conditions.	
C9H	20H	RTM_RETIRED.ABORTED_UNFRIENDLY	Number of times an RTM execution aborted due to HLE-unfriendly instructions.	
C9H	40H	RTM_RETIRED.ABORTED_MEMTYPE	Number of times an RTM execution aborted due to incompatible memory type.	
C9H	80H	RTM_RETIRED.ABORTED_EVENTS	Number of times an RTM execution aborted due to none of the previous 4 categories (for example, interrupt).	

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Haswell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3CH and 06_45H support performance events listed in Table 19-10.

Table 19-10. Non-Architectural Uncore Performance Events in the 4th Generation Intel® Core™ Processors

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to L3 eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	L3 lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	06H	UNC_CBO_CACHE_LOOKUP.ES	L3 lookup request that access cache and found line in E or S state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	L3 lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	

Table 19-10. Non-Architectural Uncore Performance Events in the 4th Generation Intel® Core™ Processors (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or L3.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or L3.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and L3 evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of L3 evictions allocated.	
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.5.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v3 Family

Non-architectural performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v3 family based on the Haswell-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3FH, are listed in Table 19-11. The performance events listed in Table 19-8 and Table 19-9 also apply Intel Xeon processor E5 v3 family, except that the OFF_CORE_RESPONSE_x event listed in Table 19-8 should reference Table 18-50.

Uncore performance monitoring events for Intel Xeon Processor E5 v3 families are described in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual".

Table 19-11. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 v3 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	04H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data sources were remote DRAM (snoop not needed, Snoop Miss).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote cache HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.6 PERFORMANCE MONITORING EVENTS FOR 3RD GENERATION INTEL® CORE™ PROCESSORS

3rd generation Intel® Core™ processors and Intel Xeon processor E3-1200 v2 product family are based on Intel microarchitecture code name Ivy Bridge. They support architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-12. The events in Table 19-12 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3AH. Fixed counters in the core PMU support the architecture events defined in Table 19-23.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	81H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size from demand loads.	
08H	82H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size by demand loads.	
08H	84H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk due to demand loads.	
08H	88H	DTLB_LOAD_MISSES.LARGE_PAGE_WALK_DURATION	Page walk for a large page completed for Demand load.	
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops adds delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SiNGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* or AVX-128 double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* or AVX-128 single precision FP scalar uops executed.	

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* or AVX-128 single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR _DOUBLE	Counts number of SSE* or AVX-128 double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
24H	01H	L2_RQSTS.DEMAND_DATA_RD_H IT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_ RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks that missed LLC.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache lines in any state.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENC E	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer Move Elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD Move Elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer Move Elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD Move Elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
5FH	04H	DTLB_LOAD_MISSES.STLB_HIT	Counts load operations that missed 1st level DTLB but hit the 2nd level.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand Code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	04H	ICACHE.IFETCH_STALL	Cycles where a code-fetch stalled due to L1 instruction-cache miss or an iTLB miss.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_EXECUTE	Cycles of dispatch stalls. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Restricted to counters 0-3 when HTT is disabled.
A3H	06H	CYCLE_ACTIVITY.STALLS_LDM_PENDING		Restricted to counters 0-3 when HTT is disabled.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFFCORE_RESPONSE_1	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	80H	OTHER_ASSISTS.WB	Number of times microcode assist is invoked by hardware upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	Supports PEBS.
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	Supports PEBS.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	Supports PEBS.
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	Supports PEBS.
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	Supports PEBS.
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PMC 3 only.
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.9.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops whose data source followed an L1 miss.	Supports PEBS.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops that missed L2, excluding unknown sources.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops whose data source is LLC miss.	Supports PEBS. Restricted to counters 0-3 when HTT is disabled.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.

Table 19-12. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data source was local memory (cross-socket snoop not needed or missed).	Supports PEBS.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or LLC HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by the MLC prefetcher.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by the MLC prefetcher.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.

19.6.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v2 Family and Intel Xeon Processor E7 v2 Family

Non-architectural performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v2 family and Intel Xeon processor E7 v2 family based on the Ivy Bridge-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3EH, are listed in Table 19-13.

Table 19-13. Non-Architectural Performance Events Applicable Only to the Processor Core of Intel® Xeon® Processor E5 v2 Family and Intel® Xeon® Processor E7 v2 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	03H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data sources were local DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	0CH	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data source was remote DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.7 PERFORMANCE MONITORING EVENTS FOR 2ND GENERATION INTEL® CORE™ I7-2XXX, INTEL® CORE™ I5-2XXX, INTEL® CORE™ I3-2XXX PROCESSOR SERIES

2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel Xeon processor E3-1200 product family are based on the Intel microarchitecture code name Sandy Bridge. They support architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-14, Table 19-15, and Table 19-16. The events in Table 19-14 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_2AH and 06_2DH. The events in Table 19-15 apply to processors with CPUID signature 06_2AH. The events in Table 19-16 apply to processors with CPUID signature 06_2DH. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Blocked loads due to store buffer blocks with unknown data.	
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	# of Split loads blocked due to resource not available.	
03H	10H	LD_BLOCKS.ALL_BLOCK	Number of cases where any load is blocked but has no DCU miss.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	08H	LD_BLOCKS_PARTIAL.ALL_STA_BLOCK	The number of times that load operations are temporarily blocked because of older stores, with addresses that are not yet known. A load operation may incur more than one block of this type.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size.	
08H	04H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears or JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0DH	40H	INT_MISC.RAT_STALL_CYCLES	Cycles RAT external stall is sent to IDQ for this thread.	
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
17H	01H	INSTS_WRITTEN_TO_IQ.INSTS	Counts the number of instructions written into the IQ every cycle.	
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Requests from L2 Hardware prefetcher that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Requests from L2 Hardware prefetcher that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Any requests from L2 Hardware prefetchers.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	04H	L2_STORE_LOCK_RQSTS.HIT_E	RFOs that hit cache lines in E state.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks from L1D to L2 cache lines that missed L2.	
28H	02H	L2_L1D_WB_RQSTS.HIT_S	Not rejected writebacks from L1D to L2 cache lines in S state.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4CH	01H	LOAD_HIT_PRE.SW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4EH	02H	HW_PRE_REQ.DL1_MISS	Hardware Prefetch requests that miss the L1D cache. A request is being counted each time it access the cache & miss it, including if a block is applicable or if hit the Fill Buffer for example.	This accounts for both L1 streamer and IP-based (IPP) HW prefetchers.
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
51H	02H	L1D.ALLOCATED_IN_M	Counts the number of allocations of modified L1D cache lines.	
51H	04H	L1D.EVICTION	Counts the number of modified lines evicted from the L1 data cache due to replacement.	
51H	08H	L1D.ALL_M_REPLACEMENT	Cache lines in M state evicted out of L1D due to Snoop HitM or dirty line replacement.	
59H	20H	PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP	Increments the number of flags-merge uops in flight each cycle. Set Cmask = 1 to count cycles.	
59H	40H	PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW	Cycles with at least one slow LEA uop allocated.	
59H	80H	PARTIAL_RAT_STALLS.MUL_SINGLE_UOP	Number of Multiply packed/scalar single precision uops allocated.	
5BH	0CH	RESOURCE_STALLS2.ALL_FL_EMPTY	Cycles stalled due to free list empty.	PMC0-3 only regardless HTT.
5BH	0FH	RESOURCE_STALLS2.ALL_PRF_CONTROL	Cycles stalled due to control structures full for physical registers.	
5BH	40H	RESOURCE_STALLS2.BOB_FULL	Cycles Allocator is stalled due Branch Order Buffer.	
5BH	4FH	RESOURCE_STALLS2.OOO_RSRC	Cycles stalled due to out of order resources full.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS busy by DSB. Set Cmask = 1 to count cycles MS is busy. Set Cmask=1 and Edge =1 to count MS activations.	Can combine Umask 08H and 10H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS is busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H and 30H.
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	41H	BR_INST_EXEC.NONTAKEN_CONDITIONAL	Not-taken macro conditional branches.	
88H	81H	BR_INST_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired conditional branches.	
88H	82H	BR_INST_EXEC.TAKEN_DIRECT_JUMP	Taken speculative and retired conditional branches excluding calls and indirects.	
88H	84H	BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired indirect branches excluding calls and returns.	
88H	88H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN	Taken speculative and retired indirect branches that are returns.	
88H	90H	BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired direct near calls.	
88H	A0H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired indirect near calls.	
88H	C1H	BR_INST_EXEC.ALL_CONDITIONAL	Speculative and retired conditional branches.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	C2H	BR_INST_EXEC.ALL_DIRECT_JUMP	Speculative and retired conditional branches excluding calls and indirects.	
88H	C4H	BR_INST_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired indirect branches excluding calls and returns.	
88H	C8H	BR_INST_EXEC.ALL_INDIRECT_NEAR_RETURN	Speculative and retired indirect branches that are returns.	
88H	D0H	BR_INST_EXEC.ALL_NEAR_CALL	Speculative and retired direct near calls.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Speculative and retired branches.	
89H	41H	BR_MISP_EXEC.NONTAKEN_CONDITIONAL	Not-taken mispredicted macro conditional branches.	
89H	81H	BR_MISP_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired mispredicted conditional branches.	
89H	84H	BR_MISP_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	88H	BR_MISP_EXEC.TAKEN_RETURN_NEAR	Taken speculative and retired mispredicted indirect branches that are returns.	
89H	90H	BR_MISP_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired mispredicted direct near calls.	
89H	A0H	BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired mispredicted indirect near calls.	
89H	C1H	BR_MISP_EXEC.ALL_CONDITIONAL	Speculative and retired mispredicted conditional branches.	
89H	C4H	BR_MISP_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	D0H	BR_MISP_EXEC.ALL_NEAR_CALL	Speculative and retired mispredicted direct near calls.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Speculative and retired mispredicted branches.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	02H	RESOURCE_STALLS.LB	Counts the cycles of stall due to lack of load buffers.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FCSW	Cycles stalled due to writing the FPU control word.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	
A3H	02H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_DISPATCH	Cycles of dispatch stalls. Set AnyThread to count per core.	PMC0-3 only.
A3H	05H	CYCLE_ACTIVITY.STALL_CYCLES_L2_PENDING		PMC0-3 only.
A3H	06H	CYCLE_ACTIVITY.STALL_CYCLES_L1D_PENDING		PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	02H	DSB_FILL.OTHER_CANCEL	Cases of cancelling valid DSB fill not because of exceeding way limit.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_DISPATCHED.THREAD	Counts total number of uops to be dispatched per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	PMC0-3 only regardless HTT.
B1H	02H	UOPS_DISPATCHED.CORE	Counts total number of uops to be dispatched per-core each cycle.	Do not need to set ANY.
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this thread core.	
B6H	01H	AGU_BYPASS_CANCEL.COUNT	Counts executed load operations with all the following traits: 1. Addressing of the format [base + offset], 2. The offset is between 1 and 2047, 3. The address specified in the base register is in one page and the address [base+offset] is in another page.	

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.9.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
BFH	05H	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports.	Cmask=1.
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only; must quiesce other PMCs.
C1H	02H	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.	
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	10H	BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 assists due to output value.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 assists due to input value.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization. PMC3 only.	Specify threshold in MSR 3F6H.
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.9.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.

Table 19-14. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops which data sources were data hits in LLC without snoops required.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops which data sources were data missed LLC (excluding unknown data source).	Supports PEBS.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
E6H	01H	BACLEARS.ANY	Counts the number of times the front end is re-steered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front end.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	L2 or LLC HW prefetches that access L2 cache.	Including rejects.
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by L2 prefetch.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by L2 prefetch.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.
F4H	10H	SQ_MISC.SPLIT_LOCK	Split locks in SQ.	

Non-architecture performance monitoring events in the processor core that are applicable only to Intel processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH are listed in Table 19-15.

Table 19-15. Non-Architectural Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were LLC hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS. PMC0-3 only regardless HTT.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were LLC and cross-core snoop hits in on-pkg core cache.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared LLC.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in LLC without snoops required.	Supports PEBS.
D4H	02H	MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS	Retired load uops with unknown information as data source in cache serviced the load.	Supports PEBS. PMC0-3 only regardless HTT.
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT_N		10003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.MISS_DRAM_N		300400244H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0091H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_MISS.DRAM_N		300400091H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_MISS.DRAM_N		300400240H
		OFFCORE_RESPONSE.ALL_PF_DATA_RD.LLC_MISS.DRAM_N		300400090H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0120H
		OFFCORE_RESPONSE.ALL_PF_RfO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.SNOOP_MISS_N		2003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_MISS.DRAM_N		300400120H
		OFFCORE_RESPONSE.ALL_READS.LLC_MISS.DRAM_N		3004003F7H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.SNOOP_MISS_N		2003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_MISS.DRAM_N		300400122H

Table 19-15. Non-Architectural Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.DRAM_N		300400004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.DRAM_N		300400001H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.SNOOP_MISS_N		2003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_MISS.DRAM_N		300400002H
		OFFCORE_RESPONSE.OTHER.ANY_RESPONSE_N		18000H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.DRAM_N		300400040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.DRAM_N		300400010H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.SNOOP_MISS_N		2003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_MISS.DRAM_N		300400020H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.DRAM_N		300400200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.DRAM_N		300400080H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.SNOOP_MISS_N		2003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_MISS.DRAM_N		300400100H

Non-architecture performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 family (and Intel Core i7-3930 processor) based on Intel microarchitecture code name Sandy Bridge, with CPUID signature of DisplayFamily_DisplayModel 06_2DH, are listed in Table 19-16.

Table 19-16. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Additional Configuration: Disable BL bypass and direct2core, and if the memory is remotely homed. The count is not reliable If the memory is locally homed.	
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Additional Configuration: Disable BL bypass. Supports PEBS.	
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Additional Configuration: Disable BL bypass and direct2core. Supports PEBS.	
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Additional Configuration: Disable bypass. Supports PEBS.	
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources were data missed LLC but serviced by local DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
D3H	04H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops which data sources were data missed LLC but serviced by remote DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.LOCAL_DRAM_N		600400004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_DRAM_N		67F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HITM_N		107FC00004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00001H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00010H

Table 19-16. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00080H

Non-architectural Performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Sandy Bridge. Processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH support performance events listed in Table 19-17.

Table 19-17. Non-Architectural Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to LLC eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	LLC lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	02H	UNC_CBO_CACHE_LOOKUP.E	LLC lookup request that access cache and found line in E-state.	
34H	04H	UNC_CBO_CACHE_LOOKUP.S	LLC lookup request that access cache and found line in S-state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	LLC lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	

Table 19-17. Non-Architectural Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or LLC.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or LLC.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and LLC evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of LLC evictions allocated.	
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.8 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ I7 PROCESSOR FAMILY AND INTEL® XEON® PROCESSOR FAMILY

Processors based on the Intel microarchitecture code name Nehalem support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-18. The events in Table 19-18 generally applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_1AH, 06_1EH, 06_1FH, and 06_2EH. However, Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a small number of events that are not supported in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. These events are noted in the comment column.

In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, 06_1FH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table 19-19.

Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	07H	SB_DRAIN.ANY	Counts the number of store buffer drains.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
08H	80H	DTLB_LOAD_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to load miss in the STLB.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set "invert=1, cmask = 1".
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	
0FH	01H	MEM_UNCORE_RETIRED.L3_DATA_MISS_UNKNOWN	Counts number of memory load instructions retired where the memory reference missed L3 and data source is unknown.	Available only for CPUID signature 06_2EH.
0FH	02H	MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM	Counts number of memory load instructions retired where the memory reference hit modified data in a sibling core residing on the same socket.	
0FH	08H	MEM_UNCORE_RETIRED.REMOTE_CACHE_LOCAL_HOME_HIT	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and HIT in a remote socket's cache. Only counts locally homed lines.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and was remotely homed. This includes both DRAM access and HITM in a remote socket's cache for remotely homed lines.	
0FH	20H	MEM_UNCORE_RETIRED.LOCAL_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and required a local socket memory reference. This includes locally homed cachelines that were in a modified state in another socket.	
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and to perform I/O.	Available only for CPUID signature 06_2EH.
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Counts number of loops that can't stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, for example, a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	
2EH	4FH	L3_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache. The event count includes speculative traffic but excludes cache line fills due to a L2 hardware-prefetch. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	41H	L3_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache. The event count may include speculative traffic but excludes cache line fills due to L2 hardware-prefetches. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
40H	01H	L1D_CACHE_LD.I_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the I (invalid) state, i.e. the read request missed the cache.	Counter 0, 1 only.
40H	02H	L1D_CACHE_LD.S_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.
40H	04H	L1D_CACHE_LD.E_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
40H	08H	L1D_CACHE_LD.M_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
40H	0FH	L1D_CACHE_LD.MESI	Counts L1 data cache read requests.	Counter 0, 1 only.
41H	02H	L1D_CACHE_ST.S_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	04H	L1D_CACHE_ST.E_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
41H	08H	L1D_CACHE_ST.M_STATE	Counts L1 data cache store RFO requests where cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
42H	01H	L1D_CACHE_LOCK.HIT	Counts retired load locks that hit in the L1 data cache or hit in an already allocated fill buffer. The lock portion of the load lock transaction must hit in the L1D.	The initial load will pull the lock into the L1 data cache. Counter 0, 1 only.
42H	02H	L1D_CACHE_LOCK.S_STATE	Counts L1 data cache retired load locks that hit the target cache line in the shared state.	Counter 0, 1 only.
42H	04H	L1D_CACHE_LOCK.E_STATE	Counts L1 data cache retired load locks that hit the target cache line in the exclusive state.	Counter 0, 1 only.
42H	08H	L1D_CACHE_LOCK.M_STATE	Counts L1 data cache retired load locks that hit the target cache line in the modified state.	Counter 0, 1 only.
43H	01H	L1D_ALL_REF.ANY	Counts all references (uncached, speculated and retired) to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once.	The event does not include non-memory accesses, such as I/O accesses. Counter 0, 1 only.
43H	02H	L1D_ALL_REF.CACHEABLE	Counts all data reads and writes (speculated and retired) from cacheable memory, including locked operations.	Counter 0, 1 only.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk for large pages.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
53H	01H	L1D_CACHE_LOCK_FB_HIT	Counts the number of cacheable load lock speculated or retired instructions accepted into the fill buffer.	
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts cycles when the uops executed were issued from any ports except port 5. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts cycles when the uops are executing. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.8.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.8.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere".	Requires programming MSR 01A7H.
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of MMX instructions retired.	
C0H	04H	INST_RETIRED.MMX	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating point Uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating point Uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating point Uops retired.	
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating point Uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer Uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_POR T	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe. Cycles when partial register stalls occurred. Cycles when flag stalls occurred. Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating-point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions (denormal input when the DAZ flag is off or underflow result when the FTZ flag is off); x87 instructions (NaN or denormal are loaded to a register or used as input from memory, division by 0 or underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	

Table 19-18. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Nehalem. Processors with CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, and 06_1FH support performance events listed in Table 19-19.

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
20H	01H	UNC_QHL_REQUESTS.IOH_RE ADS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WR ITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE _READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE _WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_ READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_ WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
27H	01H	UNC_QMC_NORMAL_FULL.READ.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with read requests.	
27H	02H	UNC_QMC_NORMAL_FULL.READ.CH1	Uncore cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with read requests.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
27H	04H	UNC_QMC_NORMAL_FULL.READ.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with read requests.	
27H	08H	UNC_QMC_NORMAL_FULL.WRITE.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with write requests.	
27H	10H	UNC_QMC_NORMAL_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with write requests.	
27H	20H	UNC_QMC_NORMAL_FULL.WRITE.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with write requests.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTOPRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTOPRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	
63H	20H	UNC_DRAM_READ_CAS.AUTOPRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	
64H	02H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	
64H	20H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	

Table 19-19. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	

Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a distinct uncore sub-system that is significantly different from the uncore found in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. Non-architectural Performance monitoring events for its uncore will be available in future documentation.

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

Intel 64 processors based on Intel® microarchitecture code name Westmere support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-20. Table 19-20 applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_25H, 06_2CH. In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table 19-21. Fixed counters support the architecture events defined in Table 19-2.

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LOAD_BLOCK.OVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAIN.ANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORY.STORE	All store referenced with misaligned address.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	04H	DTLB_LOAD_MISSES.WALK_CYCLES	Cycles PMH is busy with a page walk due to a load miss in the STLB.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set "invert=1, cmask = 1".
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	
0FH	01H	MEM_UNCORE_RETIRED.UNKNOWN_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets.
0FH	02H	MEM_UNCORE_RETIRED.OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets.
0FH	04H	MEM_UNCORE_RETIRED.REMOTE_HITM	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only.
0FH	08H	MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets.
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM (Precise Event).	Applicable to two sockets only.
0FH	20H	MEM_UNCORE_RETIRED.OTHER_LLC_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only.
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets.

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Number of loops that cannot stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2EH	41H	L3_LAT_CACHE.MISS	Counts uncore Last Level Cache misses. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	4FH	L3_LAT_CACHE.REFERENCE	Counts uncore Last Level Cache references. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	04H	DTLB_MISSES.WALK_CYCLES	Counts cycles of page walk due to misses in the STLB.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	Counter 0, 1 only.
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	Counter 0, 1 only.
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	Counter 0, 1 only.

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	Counter 0, 1 only.
4FH	10H	EPT.WALK_CYCLES	Counts Extended Page walk cycles.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_DATA	Counts weighted cycles of offcore demand data read requests. Does not include L2 prefetch requests.	Counter 0.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_CODE	Counts weighted cycles of offcore demand code read requests. Does not include L2 prefetch requests.	Counter 0.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.RFO	Counts weighted cycles of offcore demand RFO requests. Does not include L2 prefetch requests.	Counter 0.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ANY.READ	Counts weighted cycles of offcore read requests of any kind. Include L2 prefetch requests.	Counter 0.
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table. This event does not cause locks, it merely detects them.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
85H	04H	ITLB_MISSES.WALK_CYCLES	Counts ITLB miss page walk cycles.	
85H	10H	ITLB_MISSES.STLB_HIT	Counts number of ITLB first level miss but second level hits.	
85H	80H	ITLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	
B0H	01H	OFFCORE_REQUESTS.DEMAND.READ_DATA	Counts number of offcore demand data read requests. Does not count L2 prefetch requests.	
B0H	02H	OFFCORE_REQUESTS.DEMAND.READ_CODE	Counts number of offcore demand code read requests. Does not count L2 prefetch requests.	
B0H	04H	OFFCORE_REQUESTS.DEMAND.RFO	Counts number of offcore demand RFO requests. Does not count L2 prefetch requests.	
B0H	08H	OFFCORE_REQUESTS.ANY.READ	Counts number of offcore read requests. Includes L2 prefetch requests.	
B0H	10H	OFFCORE_REQUESTS.ANY.RFO	Counts number of offcore RFO requests. Includes L2 prefetch requests.	
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B0H	80H	OFFCORE_REQUESTS.ANY	Counts all offcore requests.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts number of cycles there are one or more uops being executed and were issued on ports 0-4. This is a core count only and cannot be collected per thread.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts number of cycles there are one or more uops being executed on any ports. This is a core count only and cannot be collected per thread.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B3H	01H	SNOOPQ_REQUESTS_OUTSTANDING.DATA	Counts weighted cycles of snoopq requests for data. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	02H	SNOOPQ_REQUESTS_OUTSTANDING.INVALIDATE	Counts weighted cycles of snoopq invalidate requests. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	04H	SNOOPQ_REQUESTS_OUTSTANDING.CODE	Counts weighted cycles of snoopq requests for code. Counter 0 only.	Use cmask=1 to count cycles not empty.
B4H	01H	SNOOPQ_REQUESTS.CODE	Counts the number of snoop code requests.	
B4H	02H	SNOOPQ_REQUESTS.DATA	Counts the number of snoop data requests.	
B4H	04H	SNOOPQ_REQUESTS.INVALIDATE	Counts the number of snoop invalidate requests.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.8.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.8.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Use MSR 01A7H.

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C0H	04H	INST_RETIRED.MMX	Counts the number of retired: MMX instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Counts mispredicted conditional retired calls.	
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Counts all mispredicted retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating-point uops retired.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating-point uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating-point uops retired.	
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating-point uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	01H	UOPS_DECODED.STALL_CYCLES	Counts the cycles of decoder stalls. INV=1, Cmask= 1.	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_PORT	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe, Cycles when partial register stalls occurred, Cycles when flag stalls occurred, Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
ECH	01H	THREAD_ACTIVE	Counts cycles threads are active.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	04H	SQ_MISC.LRU_HINTS	Counts number of Super Queue LRU hints sent to L3.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions, (Denormal input when the DAZ flag is off or Underflow result when the FTZ flag is off): x87 instructions, (NaN or denormal are loaded to a register or used as input from memory, Division by 0 or Underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	

Table 19-20. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Non-architectural Performance monitoring events of the uncore sub-system for processors with CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH, and 06_1FH support performance events listed in Table 19-21.

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
02H	01H	UNC_GQ_OCCUPANCY.READ_TRACKER	Increments the number of queue entries (code read, data read, and RFOs) in the tread tracker. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
0CH	01H	UNC_GQ_SNOOP.GOTO_S	Counts the number of remote snoops that have requested a cache line be set to the S state.	
0CH	02H	UNC_GQ_SNOOP.GOTO_I	Counts the number of remote snoops that have requested a cache line be set to the I state.	
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the S state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the S state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the S state from M state.	Requires writing MSR 301H with mask = 1H.

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the S state from S state.	Requires writing MSR 301H with mask = 4H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the I state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the I state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the I state from M state.	Requires writing MSR 301H with mask = 1H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the I state from S state.	Requires writing MSR 301H with mask = 4H.
20H	01H	UNC_QHL_REQUESTS.IOH_READS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WRITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE_READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE_WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2AH	07H	UNC_QMC_OCCUPANCY.ANY	Normal read request occupancy for any channel.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
32H	01H	UNC_IMC_RETRY.CH0	Counts number of IMC DRAM channel 0 retries. DRAM retry only occurs when configured in RAS mode.	
32H	02H	UNC_IMC_RETRY.CH1	Counts number of IMC DRAM channel 1 retries. DRAM retry only occurs when configured in RAS mode.	
32H	04H	UNC_IMC_RETRY.CH2	Counts number of IMC DRAM channel 2 retries. DRAM retry only occurs when configured in RAS mode.	
32H	07H	UNC_IMC_RETRY.ANY	Counts number of IMC DRAM retries from any channel. DRAM retry only occurs when configured in RAS mode.	
33H	01H	UNC_QHL_FRC_ACK_CNFLTS.IOH	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the IOH.	
33H	02H	UNC_QHL_FRC_ACK_CNFLTS.REMOTE	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the remote home.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	
33H	07H	UNC_QHL_FRC_ACK_CNFLTS.ANY	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic.	
34H	01H	UNC_QHL_SLEEPS.IOH_ORDER	Counts number of occurrences a request was put to sleep due to IOH ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	02H	UNC_QHL_SLEEPS.REMOTE_ORDER	Counts number of occurrences a request was put to sleep due to remote socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	04H	UNC_QHL_SLEEPS.LOCAL_ORDER	Counts number of occurrences a request was put to sleep due to local socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	08H	UNC_QHL_SLEEPS.IOH_CONFLICT	Counts number of occurrences a request was put to sleep due to IOH address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
34H	10H	UNC_QHL_SLEEPS.REMOTE_CONFLICT	Counts number of occurrences a request was put to sleep due to remote socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	20H	UNC_QHL_SLEEPS.LOCAL_CONFLICT	Counts number of occurrences a request was put to sleep due to local socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
35H	01H	UNC_ADDR_OPCODE_MATCH.IOH	Counts number of requests from the IOH, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	02H	UNC_ADDR_OPCODE_MATCH.REMOTE	Counts number of requests from the remote socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	04H	UNC_ADDR_OPCODE_MATCH.LOCAL	Counts number of requests from the local socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	01H	UNC_QPI_TX_HEADER.FULL.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is full.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	04H	UNC_QPI_TX_HEADER.FULL.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is full.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTOPRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTOPRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	
63H	20H	UNC_DRAM_READ_CAS.AUTOPRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
64H	02H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	
64H	20H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
67H	01H	UNC_DRAM_THERMAL_THROTTLED	Uncore cycles DRAM was throttled due to its temperature being above the thermal throttling threshold.	
80H	01H	UNC_THERMAL_THROTTLING_TEMP.CORE_0	Cycles that the PCU records that core 0 is above the thermal throttling threshold temperature.	
80H	02H	UNC_THERMAL_THROTTLING_TEMP.CORE_1	Cycles that the PCU records that core 1 is above the thermal throttling threshold temperature.	
80H	04H	UNC_THERMAL_THROTTLING_TEMP.CORE_2	Cycles that the PCU records that core 2 is above the thermal throttling threshold temperature.	
80H	08H	UNC_THERMAL_THROTTLING_TEMP.CORE_3	Cycles that the PCU records that core 3 is above the thermal throttling threshold temperature.	

Table 19-21. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
81H	01H	UNC_THERMAL_THROTTLED_TEMP.CORE_0	Cycles that the PCU records that core 0 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	02H	UNC_THERMAL_THROTTLED_TEMP.CORE_1	Cycles that the PCU records that core 1 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	04H	UNC_THERMAL_THROTTLED_TEMP.CORE_2	Cycles that the PCU records that core 2 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	08H	UNC_THERMAL_THROTTLED_TEMP.CORE_3	Cycles that the PCU records that core 3 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
82H	01H	UNC_PROCHOT_ASSERTION	Number of system assertions of PROCHOT indicating the entire processor has exceeded the thermal limit.	
83H	01H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_0	Cycles that the PCU records that core 0 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	02H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_1	Cycles that the PCU records that core 1 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	04H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_2	Cycles that the PCU records that core 2 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	08H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_3	Cycles that the PCU records that core 3 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
84H	01H	UNC_TURBO_MODE.CORE_0	Uncore cycles that core 0 is operating in turbo mode.	
84H	02H	UNC_TURBO_MODE.CORE_1	Uncore cycles that core 1 is operating in turbo mode.	
84H	04H	UNC_TURBO_MODE.CORE_2	Uncore cycles that core 2 is operating in turbo mode.	
84H	08H	UNC_TURBO_MODE.CORE_3	Uncore cycles that core 3 is operating in turbo mode.	
85H	02H	UNC_CYCLES_UNHALTED_L3_FLL_ENABLE	Uncore cycles that at least one core is unhalted and all L3 ways are enabled.	
86H	01H	UNC_CYCLES_UNHALTED_L3_FLL_DISABLE	Uncore cycles that at least one core is unhalted and all L3 ways are disabled.	

19.10 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 5200, 5400 SERIES AND INTEL® CORE™2 EXTREME PROCESSORS QX 9000 SERIES

Processors based on the Enhanced Intel Core microarchitecture support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-24. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-22. Fixed counters support the architecture events defined in Table 19-23.

Table 19-22. Non-Architectural Performance Events for Processors Based on Enhanced Intel Core Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	08H	INST_RETIRED.VM_HOST	Instruction retired while in VMX root operations.	
D2H	10H	RAT_STAALS.OTHER_SERIALIZATION_STALLS	This event counts the number of stalls due to other RAT resource serialization not counted by Umask value 0FH.	

19.11 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 3000, 3200, 5100, 5300 SERIES AND INTEL® CORE™2 DUO PROCESSORS

Processors based on the Intel® Core™ microarchitecture support architectural and non-architectural performance-monitoring events.

Fixed-function performance counters are introduced first on processors based on Intel Core microarchitecture. Table 19-23 lists pre-defined performance events that can be counted using fixed-function performance counters.

Table 19-23. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
MSR_PERF_FIXED_CTR1/IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.CORE	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time. When the core frequency is constant, this event can approximate elapsed time while the core was not in halt state.
MSR_PERF_FIXED_CTR2/IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF	This event counts the number of reference cycles when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in halt state and not in a TM stop-clock state. This event has a constant ratio with the CPU_CLK_UNHALTED.BUS event.

Table 19-24 lists general-purpose non-architectural performance-monitoring events supported in processors based on Intel® Core™ microarchitecture. For convenience, Table 19-24 also includes architectural events and describes minor model-specific behavior where applicable. Software must use a general-purpose performance counter to count events listed in Table 19-24.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture

Event Num	Umask Value	Event Name	Definition	Description and Comment
03H	02H	LOAD_BLOCK.STA	Loads blocked by a preceding store with unknown address.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to an address that is not yet calculated. The number of events is greater or equal to the number of load operations that were blocked. If the load and the store are always to different addresses, check why the memory disambiguation mechanism is not working. To avoid such blocks, increase the distance between the store and the following load so that the store address is known at the time the load is dispatched.
03H	04H	LOAD_BLOCK.STD	Loads blocked by a preceding store with unknown data.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to the same address and the stored data value is not yet known. The number of events is greater or equal to the number of load operations that were blocked. To avoid such blocks, increase the distance between the store and the dependent load, so that the store data is known at the time the load is dispatched.
03H	08H	LOAD_BLOCK. OVERLAP_STORE	Loads that partially overlap an earlier store, or 4-Kbyte aliased with a previous store.	This event indicates that loads are blocked due to a variety of reasons. Some of the triggers for this event are when a load is blocked by a preceding store, in one of the following: • Some of the loaded byte locations are written by the preceding store and some are not. • The load is from bytes written by the preceding store, the store is aligned to its size and either: • The load's data size is one or two bytes and it is not aligned to the store. • The load's data size is of four or eight bytes and the load is misaligned. • The load is from bytes written by the preceding store, the store is misaligned and the load is not aligned on the beginning of the store. • The load is split over an eight byte boundary (excluding 16-byte loads). • The load and store have the same offset relative to the beginning of different 4-KByte pages. This case is also called 4-KByte aliasing. • In all these cases the load is blocked until after the blocking store retires and the stored data is committed to the cache hierarchy.
03H	10H	LOAD_BLOCK. UNTIL_RETIRE	Loads blocked until retirement.	This event indicates that load operations were blocked until retirement. The number of events is greater or equal to the number of load operations that were blocked. This includes mainly uncacheable loads and split loads (loads that cross the cache line boundary) but may include other cases where loads are blocked until retirement.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
03H	20H	LOAD_BLOCK.L1D	Loads blocked by the L1 data cache.	This event indicates that loads are blocked due to one or more reasons. Some triggers for this event are: ▪ The number of L1 data cache misses exceeds the maximum number of outstanding misses supported by the processor. This includes misses generated as result of demand fetches, software prefetches or hardware prefetches. ▪ Cache line split loads. ▪ Partial reads, such as reads to un-cacheable memory, I/O instructions and more. ▪ A locked load operation is in progress. The number of events is greater or equal to the number of load operations that were blocked.
04H	01H	SB_DRAIN_ CYCLES	Cycles while stores are blocked due to store buffer drain.	This event counts every cycle during which the store buffer is draining. This includes: ▪ Serializing operations such as CPUID ▪ Synchronizing operations such as XCHG ▪ Interrupt acknowledgment ▪ Other conditions, such as cache flushing
04H	02H	STORE_BLOCK. ORDER	Cycles while store is waiting for a preceding store to be globally observed.	This event counts the total duration, in number of cycles, which stores are waiting for a preceding stored cache line to be observed by other cores. This situation happens as a result of the strong store ordering behavior, as defined in "Memory Ordering," Chapter 8, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. The stall may occur and be noticeable if there are many cases when a store either misses the L1 data cache or hits a cache line in the Shared state. If the store requires a bus transaction to read the cache line then the stall ends when snoop response for the bus transaction arrives.
04H	08H	STORE_BLOCK. SNOOP	A store is blocked due to a conflict with an external or internal snoop.	This event counts the number of cycles the store port was used for snooping the L1 data cache and a store was stalled by the snoop. The store is typically resubmitted one cycle later.
06H	00H	SEGMENT_REG_ LOADS	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
07H	00H	SSE_PRE_EXEC. NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
07H	01H	SSE_PRE_EXEC.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	02H	SSE_PRE_EXEC.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.
07H	03H	SSE_PRE_ EXEC.STORES	Streaming SIMD Extensions (SSE) Weakly-ordered store instructions executed.	This event counts the number of times SSE non-temporal store instructions are executed.
08H	01H	DTLB_MISSES. ANY	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	02H	DTLB_MISSES .MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	04H	DTLB_MISSES.L0_MISS_LD	L0 DTLB misses due to load operations.	This event counts the number of level 0 Data Table Lookaside Buffer (DTLB0) misses due to load operations. This count includes misses detected as a result of speculative accesses. Loads that miss that DTLB0 and hit the DTLB1 can incur two-cycle penalty.
08H	08H	DTLB_MISSES. MISS_ST	TLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses. Address translation for store operations is performed in the DTLB1.
09H	01H	MEMORY_ DISAMBIGUATION.RESET	Memory disambiguation reset cycles.	This event counts the number of cycles during which memory disambiguation misprediction occurs. As a result the execution pipeline is cleaned and execution of the mispredicted load instruction and all succeeding instructions restarts. This event occurs when the data address accessed by a load instruction, collides infrequently with preceding stores, but usually there is no collision. It happens rarely, and may have a penalty of about 20 cycles.
09H	02H	MEMORY_DISAMBIGUATIO N.SUCCESS	Number of loads successfully disambiguated.	This event counts the number of load operations that were successfully disambiguated. Loads are preceded by a store with an unknown address, but they are not blocked.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
0CH	01H	PAGE_WALKS .COUNT	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. The average can hint whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
0CH	02H	PAGE_WALKS. CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. The average can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
10H	00H	FP_COMP_OPS _EXE	Floating point computational micro-ops executed.	This event counts the number of floating point computational micro-ops executed. Use IA32_PMC0 only.
11H	00H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: ▪ Streaming SIMD Extensions (SSE) instructions: ▪ Denormal input when the DAZ (Denormals Are Zeros) flag is off ▪ Underflow result when the FTZ (Flush To Zero) flag is off ▪ X87 instructions: ▪ NaN or denormal are loaded to a register or used as input from memory ▪ Division by 0 ▪ Underflow output Use IA32_PMC1 only.
12H	00H	MUL	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations. Use IA32_PMC1 only.
13H	00H	DIV	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed. Use IA32_PMC1 only.
14H	00H	CYCLES_DIV _BUSY	Cycles the divider busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Use IA32_PMC0 only.
18H	00H	IDLE_DURING _DIV	Cycles the divider is busy and all other execution units are idle.	This event counts the number of cycles the divider is busy (with a divide or a square root operation) and no other execution unit or load operation is in progress. Load operations are assumed to hit the L1 data cache. This event considers only micro-ops dispatched after the divider started operating. Use IA32_PMC0 only.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
19H	00H	DELAYED_ BYPASS.FP	Delayed bypass to FP operation.	This event counts the number of times floating point operations use data immediately after the data was generated by a non-floating point execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	01H	DELAYED_ BYPASS.SIMD	Delayed bypass to SIMD operation.	This event counts the number of times SIMD operations use data immediately after the data was generated by a non-SIMD execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	02H	DELAYED_ BYPASS.LOAD	Delayed bypass to load operation.	This event counts the number of delayed bypass penalty cycles that a load operation incurred. When load operations use data immediately after the data was generated by an integer execution unit, they may (pending on certain dynamic internal conditions) incur one penalty cycle due to delayed data bypass between the units. Use IA32_PMC1 only.
21H	See Table 18-3	L2_ADS.(Core)	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. It can count occurrences for this core or both cores.
23H	See Table 18-3	L2_DBUS_BUSY _RD.(Core)	Cycles the L2 transfers data to the core.	This event counts the number of cycles during which the L2 data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. This event can count occurrences for this core or both cores.
24H	Combined mask from Table 18-3 and Table 18-5	L2_LINES_IN. (Core, Prefetch)	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. It can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-3	L2_M_LINES_IN. (Core)	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.
26H	See Table 18-3 and Table 18-5	L2_LINES_OUT. (Core, Prefetch)	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-3 and Table 18-5	L2_M_LINES_OUT.(Core, Prefetch)	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a modified-state in one of the L1 data caches. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
28H	Com-bined mask from Table 18-3 and Table 18-6	L2_IFETCH.(Core, Cache Line State)	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the IFU. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
29H	Combined mask from Table 18-3, Table 18-5, and Table 18-6	L2_LD.(Core, Prefetch, Cache Line State)	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. The event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together or separately. ▪ Of accesses to cache lines at different MESI states.
2AH	See Table 18-3 and Table 18-6	L2_ST.(Core, Cache Line State)	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
2BH	See Table 18-3 and Table 18-6	L2_LOCK.(Core, Cache Line State)	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
2EH	See Table 18-3, Table 18-5, and Table 18-6	L2_RQSTS.(Core, Prefetch, Cache Line State)	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
2EH	41H	L2_RQSTS.SELF.DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF.DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
30H	See Table 18-3, Table 18-5, and Table 18-6	L2_REJECT_BUSQ.(Core, Prefetch, Cache Line State)	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: ▪ The bus queue is full. ▪ The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
32H	See Table 18-3	L2_NO_REQ.(Core)	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests were pending from a core. When using the BOTH_CORE modifier, the event counts only if none of the cores have a pending request. The event counts also when one core is halted and the other is not halted. The event can count occurrences for this core or both cores.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep Technology (EIST) transitions.	This event counts the number of transitions that include a frequency change, either with or without voltage change. This includes Enhanced Intel SpeedStep Technology (EIST) and TM2 transitions. The event is incremented only while the counting core is in C0 state. Since transitions to higher-numbered CxE states and TM2 transitions include a frequency change or voltage transition, the event is incremented accordingly.
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond and returns to normal when the temperature falls below the thermal trip threshold temperature.
3CH	00H	CPU_CLK_ UNHALTED. CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason, this event may have a changing ratio in regard to time. When the core frequency is constant, this event can give approximate elapsed time while the core not in halt state. This is an architectural performance event.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
3CH	01H	CPU_CLK_ UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give a measurement of the elapsed time while the core was not in the halt state. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_ UNHALTED.NO _OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	See Table 18-6	L1D_CACHE_LD. (Cache Line State)	L1 cacheable data reads.	This event counts the number of data reads from cacheable memory. Locked reads are not counted.
41H	See Table 18-6	L1D_CACHE_ST. (Cache Line State)	L1 cacheable data writes.	This event counts the number of data writes to cacheable memory. Locked writes are not counted.
42H	See Table 18-6	L1D_CACHE_ LOCK.(Cache Line State)	L1 data cacheable locked reads.	This event counts the number of locked data reads from cacheable memory.
42H	10H	L1D_CACHE_ LOCK_DURATION	Duration of L1 data cacheable locked operation.	This event counts the number of cycles during which any cache line is locked by any locking instruction. Locking happens at retirement and therefore the event does not occur for instructions that are speculatively executed. Locking duration is shorter than locked instruction execution duration.
43H	01H	L1D_ALL_REF	All references to the L1 data cache.	This event counts all references to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once. The event includes non-cacheable accesses, such as I/O accesses.
43H	02H	L1D_ALL_ CACHE_REF	L1 Data cacheable reads and writes.	This event counts the number of data reads and writes from cacheable memory, including locked operations. This event is a sum of: ▪ L1D_CACHE_LD.MESI ▪ L1D_CACHE_ST.MESI ▪ L1D_CACHE_LOCK.MESI
45H	0FH	L1D_REPL	Cache lines allocated in the L1 data cache.	This event counts the number of lines brought into the L1 data cache.
46H	00H	L1D_M_REPL	Modified cache lines allocated in the L1 data cache.	This event counts the number of modified lines brought into the L1 data cache.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
47H	00H	L1D_M_EVICT	Modified cache lines evicted from the L1 data cache.	This event counts the number of modified lines evicted from the L1 data cache, whether due to replacement or by snoop HITM intervention.
48H	00H	L1D_PEND_ MISS	Total number of outstanding L1 data cache misses at any cycle.	This event counts the number of outstanding L1 data cache misses at any cycle. An L1 data cache miss is outstanding from the cycle on which the miss is determined until the first chunk of data is available. This event counts: ▪ All cacheable demand requests. ▪ L1 data cache hardware prefetch requests. ▪ Requests to write through memory. ▪ Requests to write combine memory. Uncacheable requests are not counted. The count of this event divided by the number of L1 data cache misses, L1D_REPL, is the average duration in core cycles of an L1 data cache miss.
49H	01H	L1D_SPLIT.LOADS	Cache line split loads from the L1 data cache.	This event counts the number of load operations that span two cache lines. Such load operations are also called split loads. Split load operations are executed at retirement.
49H	02H	L1D_SPLIT. STORES	Cache line split stores to the L1 data cache.	This event counts the number of store operations that span two cache lines.
4BH	00H	SSE_PRE_ MISS.NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchNTA were executed and missed all cache levels. Due to speculation an executed instruction might not retire. This instruction prefetches the data to the L1 data cache.
4BH	01H	SSE_PRE_ MISS.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT0 were executed and missed all cache levels. Due to speculation executed instruction might not retire. The prefetchT0 instruction prefetches data to the L2 cache and L1 data cache.
4BH	02H	SSE_PRE_ MISS.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 were executed and missed all cache levels. Due to speculation, an executed instruction might not retire. The prefetchT1 and PrefetchNT2 instructions prefetch data to the L2 cache.
4CH	00H	LOAD_HIT_PRE	Load operations conflicting with a software prefetch to the same address.	This event counts load operations sent to the L1 data cache while a previous Streaming SIMD Extensions (SSE) prefetch instruction to the same cache line has started prefetching but has not yet finished.
4EH	10H	L1D_PREFETCH. REQUESTS	L1 data cache prefetch requests.	This event counts the number of times the L1 data cache requested to prefetch a data cache line. Requests can be rejected when the L2 cache is busy and resubmitted later or lost. All requests are counted, including those that are rejected.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
60H	See Table 18-3 and Table 18-4.	BUS_REQUEST_ OUTSTANDING. (Core and Bus Agents)	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. The event counts only full-line cacheable read requests from either the L1 data cache or the L2 prefetchers. It does not count Read for Ownership transactions, instruction byte fetch transactions, or any other bus transaction.
61H	See Table 18-4.	BUS_BNR_DRV. (Bus Agents)	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. To obtain the number of bus cycles during which the BNR signal is asserted, multiply the event count by two. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance.
62H	See Table 18-4.	BUS_DRDY_ CLOCKS.(Bus Agents)	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. With the 'THIS_AGENT' mask this event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. With the 'ALL_AGENTS' mask, this event counts the number of bus cycles during which any bus agent sends data on the bus. This includes all data reads and writes on the bus.
63H	See Table 18-3 and Table 18-4.	BUS_LOCK_ CLOCKS.(Core and Bus Agents)	Bus cycles when a LOCK signal asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: ▪ Uncacheable memory. ▪ Locked operation that spans two cache lines. ▪ Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses.
64H	See Table 18-3.	BUS_DATA_ RCV.(Core)	Bus cycles while processor receives data.	This event counts the number of bus cycles during which the processor is busy receiving data.
65H	See Table 18-3 and Table 18-4.	BUS_TRANS_BRD.(Core and Bus Agents)	Burst read bus transactions.	This event counts the number of burst read transactions including: ▪ L1 data cache read misses (and L1 data cache hardware prefetches). ▪ L2 hardware prefetches by the DPL and L2 streamer. ▪ IFU read misses of cacheable lines. It does not include RFO transactions.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
66H	See Table 18-3 and Table 18-4.	BUS_TRANS_RFO.(Core and Bus Agents)	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. It also counts RFO bus transactions due to locked operations.
67H	See Table 18-3 and Table 18-4.	BUS_TRANS_WB. (Core and Bus Agents)	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-3 and Table 18-4.	BUS_TRANS_ IFETCH.(Core and Bus Agents)	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-3 and Table 18-4.	BUS_TRANS_ INVAL.(Core and Bus Agents)	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: ▪ A store operation hits a shared line in the L2 cache. ▪ A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-3 and Table 18-4.	BUS_TRANS_ PWR.(Core and Bus Agents)	Partial write bus transaction.	This event counts partial write bus transactions.
6BH	See Table 18-3 and Table 18-4.	BUS_TRANS _P.(Core and Bus Agents)	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-3 and Table 18-4.	BUS_TRANS_IO.(Core and Bus Agents)	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-3 and Table 18-4.	BUS_TRANS_ DEF.(Core and Bus Agents)	Deferred bus transactions.	This event counts the number of deferred transactions.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
6EH	See Table 18-3 and Table 18-4.	BUS_TRANS_ BURST.(Core and Bus Agents)	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: ▪ Burst reads. ▪ RFOs. ▪ Explicit writebacks. ▪ Write combine lines.
6FH	See Table 18-3 and Table 18-4.	BUS_TRANS_ MEM.(Core and Bus Agents)	Memory bus transactions.	This event counts all memory bus transactions including: ▪ Burst transactions. ▪ Partial reads and writes - invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_IVAL.
70H	See Table 18-3 and Table 18-4.	BUS_TRANS_ ANY.(Core and Bus Agents)	All bus transactions.	This event counts all bus transactions. This includes: ▪ Memory transactions. ▪ IO transactions (non memory-mapped). ▪ Deferred transaction completion. ▪ Other less frequent transactions, such as interrupts.
77H	See Table 18-3 and Table 18-7.	EXT_SNOOP. (Bus Agents, Snoop Response)	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. With the 'THIS_AGENT' mask, the event counts snoop responses from this processor to bus transactions sent by this processor. With the 'ALL_AGENTS' mask the event counts all snoop responses seen on the bus.
78H	See Table 18-3 and Table 18-8.	CMP_SNOOP.(Core, Snoop Type)	L1 data cache snooped by other core.	This event counts the number of times the L1 data cache is snooped for a cache line that is needed by the other core in the same processor. The cache line is either missing in the L1 instruction or data caches of the other core, or is available for reading only and the other core wishes to write the cache line. The snoop operation may change the cache line state. If the other core issued a read request that hit this core in E state, typically the state changes to S state in this core. If the other core issued a read for ownership request (due a write miss or hit to S state) that hits this core's cache line in E or S state, this typically results in invalidation of the cache line in this core. If the snoop hits a line in M state, the state is changed at a later opportunity. These snoops are performed through the L1 data cache store port. Therefore, frequent snoops may conflict with extensive stores to the L1 data cache, which may increase store latency and impact performance.
7AH	See Table 18-4.	BUS_HIT_DRV. (Bus Agents)	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response.
7BH	See Table 18-4.	BUS_HITM_DRV. (Bus Agents)	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
7DH	See Table 18-3.	BUSQ_EMPTY. (Core)	Bus queue empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. It also counts when the core is halted and the other core is not halted. This event can count occurrences for this core or both cores.
7EH	See Table 18-3 and Table 18-4.	SNOOP_STALL_ DRV.(Core and Bus Agents)	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. To obtain the number of bus cycles during which snoops on the bus are prohibited, multiply the event count by two. During the snoop stall cycles, no new bus transactions requiring a snoop response can be initiated on the bus. A bus agent asserts a snoop stall signal if it cannot response to a snoop request within three bus cycles.
7FH	See Table 18-3.	BUS_IO_WAIT. (Core)	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. With the SELF modifier this event counts IO requests per core. With the BOTH_CORE modifier, this event increments by one for any cycle for which there is a request from either core.
80H	00H	L1I_READS	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches that bypass the Instruction Fetch Unit (IFU).
81H	00H	L1I_MISSES	Instruction Fetch Unit misses.	This event counts all instruction fetches that miss the Instruction Fetch Unit (IFU) or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	02H	ITLB.SMALL_MISS	ITLB small page misses.	This event counts the number of instruction fetches from small pages that miss the ITLB.
82H	10H	ITLB.LARGE_MISS	ITLB large page misses.	This event counts the number of instruction fetches from large pages that miss the ITLB.
82H	40H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes. This usually happens upon CR3 or CR0 writes, which are executed by the operating system during process switches.
82H	12H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches from either small or large pages that miss the ITLB.
83H	02H	INST_QUEUE.FULL	Cycles during which the instruction queue is full.	This event counts the number of cycles during which the instruction queue is full. In this situation, the core front end stops fetching more instructions. This is an indication of very long stalls in the back-end pipeline stages.
86H	00H	CYCLES_L1I_ MEM_STALLED	Cycles during which instruction fetches stalled.	This event counts the number of cycles for which an instruction fetch stalls, including stalls due to any of the following reasons: • Instruction Fetch Unit cache misses. • Instruction TLB misses. • Instruction TLB faults.
87H	00H	ILD_STALL	Instruction Length Decoder stall cycles due to a length changing prefix.	This event counts the number of cycles during which the instruction length decoder uses the slow length decoder. Usually, instruction length decoding is done in one cycle. When the slow decoder is used, instruction decoding requires 6 cycles.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				The slow decoder is used in the following cases: ▪ Operand override prefix (66H) preceding an instruction with immediate data. ▪ Address override prefix (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. To avoid instruction length decoding stalls, generate code using imm8 or imm32 values instead of imm16 values. If you must use an imm16 value, store the value in a register using “mov reg, imm32” and use the register format of the instruction.
88H	00H	BR_INST_EXEC	Branch instructions executed.	This event counts all executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.
89H	00H	BR_MISSP_EXEC	Mispredicted branch instructions executed.	This event counts the number of mispredicted branch instructions that were executed.
8AH	00H	BR_BAC_ MISSP_EXEC	Branch instructions mispredicted at decoding.	This event counts the number of branch instructions that were mispredicted at decoding.
8BH	00H	BR_CND_EXEC	Conditional branch instructions executed.	This event counts the number of conditional branch instructions executed, but not necessarily retired.
8CH	00H	BR_CND_ MISSP_EXEC	Mispredicted conditional branch instructions executed.	This event counts the number of mispredicted conditional branch instructions that were executed.
8DH	00H	BR_IND_EXEC	Indirect branch instructions executed.	This event counts the number of indirect branch instructions that were executed.
8EH	00H	BR_IND_MISSP _EXEC	Mispredicted indirect branch instructions executed.	This event counts the number of mispredicted indirect branch instructions that were executed.
8FH	00H	BR_RET_EXEC	RET instructions executed.	This event counts the number of RET instructions that were executed.
90H	00H	BR_RET_ MISSP_EXEC	Mispredicted RET instructions executed.	This event counts the number of mispredicted RET instructions that were executed.
91H	00H	BR_RET_BAC_ MISSP_EXEC	RET instructions executed mispredicted at decoding.	This event counts the number of RET instructions that were executed and were mispredicted at decoding.
92H	00H	BR_CALL_EXEC	CALL instructions executed.	This event counts the number of CALL instructions executed.
93H	00H	BR_CALL_ MISSP_EXEC	Mispredicted CALL instructions executed.	This event counts the number of mispredicted CALL instructions that were executed.
94H	00H	BR_IND_CALL_ EXEC	Indirect CALL instructions executed.	This event counts the number of indirect CALL instructions that were executed.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
97H	00H	BR_TKN_ BUBBLE_1	Branch predicted taken with bubble 1.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
98H	00H	BR_TKN_ BUBBLE_2	Branch predicted taken with bubble 2.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
A0H	00H	RS_UOPS_ DISPATCHED	Micro-ops dispatched for execution.	This event counts the number of micro-ops dispatched for execution. Up to six micro-ops can be dispatched in each cycle.
A1H	01H	RS_UOPS_ DISPATCHED.PORT0	Cycles micro-ops dispatched for execution on port 0.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Issue Ports are described in *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Use IA32_PMC0 only.
A1H	02H	RS_UOPS_ DISPATCHED.PORT1	Cycles micro-ops dispatched for execution on port 1.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	04H	RS_UOPS_ DISPATCHED.PORT2	Cycles micro-ops dispatched for execution on port 2.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	08H	RS_UOPS_ DISPATCHED.PORT3	Cycles micro-ops dispatched for execution on port 3.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	10H	RS_UOPS_ DISPATCHED.PORT4	Cycles micro-ops dispatched for execution on port 4.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	20H	RS_UOPS_ DISPATCHED.PORT5	Cycles micro-ops dispatched for execution on port 5.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
AAH	01H	MACRO_INSTS. DECODED	Instructions decoded.	This event counts the number of instructions decoded (but not necessarily executed or retired).
AAH	08H	MACRO_INSTS. CISC_DECODED	CISC Instructions decoded.	This event counts the number of complex instructions decoded. Complex instructions usually have more than four micro-ops. Only one complex instruction can be decoded at a time.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
ABH	01H	ESP.SYNCH	ESP register content synchron-ization.	This event counts the number of times that the ESP register is explicitly used in the address expression of a load or store operation, after it is implicitly used, for example by a push or a pop instruction. ESP synch micro-op uses resources from the rename pipe-stage and up to retirement. The expected ratio of this event divided by the number of ESP implicit changes is 0,2. If the ratio is higher, consider rearranging your code to avoid ESP synchronization events.
ABH	02H	ESP.ADDITIONS	ESP register automatic additions.	This event counts the number of ESP additions performed automatically by the decoder. A high count of this event is good, since each automatic addition performed by the decoder saves a micro-op from the execution units. To maximize the number of ESP additions performed automatically by the decoder, choose instructions that implicitly use the ESP, such as PUSH, POP, CALL, and RET instructions whenever possible.
B0H	00H	SIMD_UOPS_EXEC	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. It does not count MOVQ and MOVD stores from register to memory.
B1H	00H	SIMD_SAT_UOP_EXEC	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B3H	01H	SIMD_UOP_TYPE_EXEC.MUL	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHIFT	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	04H	SIMD_UOP_TYPE_EXEC.PACK	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	08H	SIMD_UOP_TYPE_EXEC.UNPACK	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	10H	SIMD_UOP_TYPE_EXEC.LOGICAL	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers. INST_RETIRED.ANY_P is an architectural performance event.
C0H	01H	INST_RETIRED.LOADS	Instructions retired, which contain a load.	This event counts the number of instructions retired that contain a load operation.
C0H	02H	INST_RETIRED.STORES	Instructions retired, which contain a store.	This event counts the number of instructions retired that contain a store operation.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C0H	04H	INST_RETIRED.OTHER	Instructions retired, with no load or store operation.	This event counts the number of instructions retired that do not contain a load or a store operation.
C1H	01H	X87_OPS_RETIRED.FXCH	FXCH instructions retired.	This event counts the number of FXCH instructions retired. Modern compilers generate more efficient code and are less likely to use this instruction. If you obtain a high count for this event consider recompiling the code.
C1H	FEH	X87_OPS_RETIRED.ANY	Retired floating-point computational operations (precise event).	This event counts the number of floating-point computational operations retired. It counts: ▪ Floating point computational operations executed by the assist handler. ▪ Sub-operations of complex floating-point instructions like transcendental instructions. This event does not count: ▪ Floating-point computational operations that cause traps or assists. ▪ Floating-point loads and stores. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C2H	01H	UOPS_RETIRED.LD_IND_BR	Fused load+op or load+indirect branch retired.	This event counts the number of retired micro-ops that fused a load with another operation. This includes: ▪ Fusion of a load and an arithmetic operation, such as with the following instruction: ADD EAX, [EBX] where the content of the memory location specified by EBX register is loaded, added to EXA register, and the result is stored in EAX. ▪ Fusion of a load and a branch in an indirect branch operation, such as with the following instructions: ▪ JMP [RDI+200] ▪ RET ▪ Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	02H	UOPS_RETIRED.STD_STA	Fused store address + data retired.	This event counts the number of store address calculations that are fused with store data emission into one micro-op. Traditionally, each store operation required two micro-ops. This event counts fusion of retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	04H	UOPS_RETIRED.MACRO_FUSION	Retired instruction pairs fused into one micro-op.	This event counts the number of times CMP or TEST instructions were fused with a conditional branch instruction into one micro-op. It counts fusion by retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code uses the processor resources more effectively.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C2H	07H	UOPS_RETIRED. FUSED	Fused micro-ops retired.	This event counts the total number of retired fused micro-ops. The counts include the following fusion types: • Fusion of load operation with an arithmetic operation or with an indirect branch (counted by event UOPS_RETIRED.LD_IND_BR) • Fusion of store address and data (counted by event UOPS_RETIRED.STD_STA) • Fusion of CMP or TEST instruction with a conditional branch instruction (counted by event UOPS_RETIRED.MACRO_FUSION) Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	08H	UOPS_RETIRED. NON_FUSED	Non-fused micro-ops retired.	This event counts the number of micro-ops retired that were not fused.
C2H	0FH	UOPS_RETIRED. ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_ NUKES.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors.
C3H	04H	MACHINE_NUKES.MEM_OR DER	Execution pipeline restart due to memory ordering conflict or memory disambiguation misprediction.	This event counts the number of times the pipeline is restarted due to either multi-threaded memory ordering conflicts or memory disambiguation misprediction. A multi-threaded memory ordering conflict occurs when a store, which is executed in another core, hits a load that is executed out of order in this core but not yet retired. As a result, the load needs to be restarted to satisfy the memory ordering model. See Chapter 8, “Multiple-Processor Management” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. To count memory disambiguation mispredictions, use the event MEMORY_DISAMBIGUATION.RESET.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. This is an architectural performance event.
C4H	01H	BR_INST_RETIRED.PRED_N OT_ TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRE D_NOT_ TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions. (precise event)	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. This is an architectural performance event.
C6H	01H	CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired SSE packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired SSE scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired SSE2 packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired SSE2 scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.
C7H	10H	SIMD_INST_RETIRED.VECTOR	Retired SSE2 vector integer instructions.	This event counts the number of SSE2 vector integer instructions retired.
C7H	1FH	SIMD_INST_RETIRED.ANY	Retired Streaming SIMD instructions (precise event).	This event counts the overall number of retired SIMD instructions that use XMM registers. To count each type of SIMD instruction separately, use the following events: • SIMD_INST_RETIRED.PACKED_SINGLE • SIMD_INST_RETIRED.SCALAR_SINGLE • SIMD_INST_RETIRED.PACKED_DOUBLE • SIMD_INST_RETIRED.SCALAR_DOUBLE • and SIMD_INST_RETIRED.VECTOR When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor.
C9H	00H	ITLB_MISS_RETIRED	Retired instructions that missed the ITLB.	This event counts the number of retired instructions that missed the ITLB when they were fetched.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CAH	01H	SIMD_COMP_ INST_RETIRED. PACKED_SINGLE	Retired computational SSE packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_ INST_RETIRED. SCALAR_SINGLE	Retired computational SSE scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_ INST_RETIRED. PACKED_DOUBLE	Retired computational SSE2 packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	08H	SIMD_COMP_INST_RETIRE D.SCALAR_DOUBLE	Retired computational SSE2 scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CBH	01H	MEM_LOAD_ RETIRED.L1D _MISS	Retired loads that miss the L1 data cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache. This includes loads from cache lines that are currently being fetched, due to a previous L1 data cache miss to the same cache line. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	02H	MEM_LOAD_ RETIRED.L1D_ LINE_MISS	L1 data cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L1 data cache and send a request to the L2 cache to fetch the missing cache line. That is the missing cache line fetching has not yet started. The event count is equal to the number of cache lines fetched from the L2 cache by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events).

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	04H	MEM_LOAD_ RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache. This event counts loads from cacheable memory only. It does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	08H	MEM_LOAD_ RETIRED.L2_LINE_MISS	L2 cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L2 cache and result in a bus request to fetch the missing cache line. That is the missing cache line fetching has not yet started. This event count is equal to the number of cache lines fetched from memory by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events). When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	10H	MEM_LOAD_ RETIRED.DTLB_ MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CCH	01H	FP_MMX_TRANS_TO_MMX	Transitions from Floating Point to MMX Instructions.	This event counts the first MMX instructions following a floating-point instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.
CCH	02H	FP_MMX_TRANS_TO_FP	Transitions from MMX Instructions to Floating Point Instructions.	This event counts the first floating-point instructions following any MMX instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed, changing the MMX state in the floating point stack.
CEH	00H	SIMD_INSTR_ RETIRED	SIMD Instructions retired.	This event counts the number of retired SIMD instructions that use MMX registers.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
D2H	01H	RAT_STALLS. ROB_READ_PORT	ROB read port stalls cycles.	This event counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read-port stall is counted again.
D2H	02H	RAT_STALLS. PARTIAL_CYCLES	Partial register stall cycles.	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction uses a register that was partially written by previous instructions.
D2H	04H	RAT_STALLS. FLAGS	Flag stall cycles.	This event counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: • An instruction modifies some, but not all, of the flags in the flag register. • The next instruction, which depends on flags, depends on flags that were not modified by this instruction.
D2H	08H	RAT_STALLS. FPSW	FPU status word stall.	This event indicates that the FPU status word (FPSW) is written. To obtain the number of times the FPSW is written divide the event count by 2. The FPSW is written by instructions with long latency; a small count may indicate a high penalty.
D2H	0FH	RAT_STALLS. ANY	All RAT stall cycles.	This event counts the number of stall cycles due to conditions described by: • RAT_STALLS.ROB_READ_PORT • RAT_STALLS.PARTIAL • RAT_STALLS.FLAGS • RAT_STALLS.FPSW.
D4H	01H	SEG_RENAME_ STALLS.ES	Segment rename stalls - ES.	This event counts the number of stalls due to the lack of renaming resources for the ES segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	02H	SEG_RENAME_ STALLS.DS	Segment rename stalls - DS.	This event counts the number of stalls due to the lack of renaming resources for the DS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
D4H	04H	SEG_RENAME_ STALLS.FS	Segment rename stalls - FS.	This event counts the number of stalls due to the lack of renaming resources for the FS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	08H	SEG_RENAME_ STALLS.GS	Segment rename stalls - GS.	This event counts the number of stalls due to the lack of renaming resources for the GS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	0FH	SEG_RENAME_ STALLS.ANY	Any (ES/DS/FS/GS) segment rename stall.	This event counts the number of stalls due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D5H	01H	SEG_REG_ RENAMES.ES	Segment renames - ES.	This event counts the number of times the ES segment register is renamed.
D5H	02H	SEG_REG_ RENAMES.DS	Segment renames - DS.	This event counts the number of times the DS segment register is renamed.
D5H	04H	SEG_REG_ RENAMES.FS	Segment renames - FS.	This event counts the number of times the FS segment register is renamed.
D5H	08H	SEG_REG_ RENAMES.GS	Segment renames - GS.	This event counts the number of times the GS segment register is renamed.
D5H	0FH	SEG_REG_ RENAMES.ANY	Any (ES/DS/FS/GS) segment rename.	This event counts the number of times any of the four segment registers (ES/DS/FS/GS) is renamed.
DCH	01H	RESOURCE_ STALLS.ROB_FULL	Cycles during which the ROB full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit the processor can handle. A high count for this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.
DCH	02H	RESOURCE_ STALLS.RS_FULL	Cycles during which the RS full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.

Table 19-24. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
DCH	04	RESOURCE_ STALLS.LD_ST	Cycles during which the pipeline has exceeded load or store limit or waiting to commit all stores.	This event counts the number of cycles while resource-related stalls occur due to: • The number of load instructions in the pipeline reached the limit the processor can handle. The stall ends when a loading instruction retires. • The number of store instructions in the pipeline reached the limit the processor can handle. The stall ends when a storing instruction commits its data to the cache or memory. • There is an instruction in the pipe that can be executed only when all previous stores complete and their data is committed in the caches or memory. For example, the SFENCE and MFENCE instructions require this behavior.
DCH	08H	RESOURCE_ STALLS.FPCW	Cycles stalled due to FPU control word write.	This event counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.
DCH	10H	RESOURCE_ STALLS.BR_MISS_CLEAR	Cycles stalled due to branch misprediction.	This event counts the number of cycles after a branch misprediction is detected at execution until the branch and all older micro-ops retire. During this time new micro-ops cannot enter the out-of-order pipeline.
DCH	1FH	RESOURCE_ STALLS.ANY	Resource related stalls.	This event counts the number of cycles while resource-related stalls occurs for any conditions described by the following events: • RESOURCE_STALLS.ROB_FULL • RESOURCE_STALLS.RS_FULL • RESOURCE_STALLS.LD_ST • RESOURCE_STALLS.FPCW • RESOURCE_STALLS.BR_MISS_CLEAR
E0H	00H	BR_INST_ DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	00H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event. This occurs mainly after task switches.
E6H	00H	BACLEARS	BACLEARS asserted.	This event counts the number of times the front end is resteered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front and. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted costs approximately 7 cycles of instruction fetch. The effect on total execution time depends on the surrounding code.
F0H	00H	PREF_RQSTS_UP	Upward prefetches issued from DPL.	This event counts the number of upward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.
F8H	00H	PREF_RQSTS_DN	Downward prefetches issued from DPL.	This event counts the number of downward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.

19.12 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE GOLDMONT MICROARCHITECTURE

Next Generation Intel Atom processors based on the Goldmont microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using a fixed counter. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-25. These events also apply to processors with CPUID signatures of 06_5CH and 06_5FH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

In Goldmont microarchitecture, performance monitoring events that support Processor Event Based Sampling (PEBS) and PEBS records that contain processor state information that are associated with at-retirement tagging are marked by "Precise Event".

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture

Event Num.	Umask Value	Event Name	Description	Comment
03H	10H	LD_BLOCKS.ALL_BLOCK	Counts anytime a load that retires is blocked for any reason.	Precise Event
03H	08H	LD_BLOCKS.UTLB_MISS	Counts loads blocked because they are unable to find their physical address in the micro TLB (UTLB).	Precise Event
03H	02H	LD_BLOCKS.STORE_FORWARD	Counts a load blocked from using a store forward because of an address/size mismatch; only one of the loads blocked from each store will be counted.	Precise Event
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Counts a load blocked from using a store forward, but did not occur because the store data was not available at the right time. The forward might occur subsequently when the data is available.	Precise Event
03H	04H	LD_BLOCKS.4K_ALIAS	Counts loads that block because their address modulo 4K matches a pending store.	Precise Event
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts every core cycle when a Data-side (walks due to data operation) page walk is in progress.	
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts every core cycle when an Instruction-side (walks due to an instruction fetch) page walk is in progress.	
05H	03H	PAGE_WALKS.CYCLES	Counts every core cycle a page-walk is in progress due to either a data memory operation, or an instruction fetch.	
0EH	00H	UOPS_ISSUED.ANY	Counts uops issued by the front end and allocated into the back end of the machine. This event counts uops that retire as well as uops that were speculatively executed but didn't retire. The sort of speculative uops that might be counted includes, but is not limited to those uops issued in the shadow of a mispredicted branch, those uops that are inserted during an assist (such as for a denormal floating-point result), and (previously allocated) uops that might be canceled during a machine clear.	
13H	02H	MISALIGN_MEM_REF.LOAD_PAGE_SPLIT	Counts when a memory load of a uop that spans a page boundary (a split) is retired.	Precise Event
13H	04H	MISALIGN_MEM_REF.STORE_PAGE_SPLIT	Counts when a memory store of a uop that spans a page boundary (a split) is retired.	Precise Event
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts memory requests originating from the core that reference a cache line in the L2 cache.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts memory requests originating from the core that miss in the L2 cache.	

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
30H	00H	L2_REJECT_XQ.ALL	Counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the intra-die interconnect (IDI) fabric. The XQ may reject transactions from the L2Q (non-cacheable requests), L2 misses and L2 write-back victims.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to ensure fairness between cores, or to delay a core's dirty eviction when the address conflicts with incoming external snoops.	
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted. This event uses a programmable general purpose performance counter.	
3CH	01H	CPU_CLK_UNHALTED.REF	Reference cycles when core is not halted. This event uses a programmable general purpose performance counter.	
51H	01H	DL1.DIRTY_EVICTION	Counts when a modified (dirty) cache line is evicted from the data L1 cache and needs to be written back to memory. No count will occur if the evicted line is clean, and hence does not require a writeback.	
80H	01H	ICACHE.HIT	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is in the Icache (hit). The event strives to count on a cache line basis, so that multiple accesses which hit in a single cache line count as one ICACHE.HIT. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	02H	ICACHE.MISSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is not in the Icache (miss). The event strives to count on a cache line basis, so that multiple accesses which miss in a single cache line count as one ICACHE.MISS. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is not in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	03H	ICACHE.ACCESSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line. The event strives to count on a cache line basis, so that multiple fetches to a single cache line count as one ICACHE.ACCESS. Specifically, the event counts when accesses from straight line code crosses the cache line boundary, or when a branch target is to a new line. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
81H	04H	ITLB.MISS	Counts the number of times the machine was unable to find a translation in the Instruction Translation Lookaside Buffer (ITLB) for a linear address of an instruction fetch. It counts when new translations are filled into the ITLB. The event is speculative in nature, but will not count translations (page walks) that are begun and not finished, or translations that are finished but not filled into the ITLB.	

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
86H	02H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts cycles that an ICache miss is outstanding, and instruction fetch is stalled. That is, the decoder queue is able to accept bytes, but the fetch unit is unable to provide bytes, while an Icache miss is outstanding. Note this event is not the same as cycles to retrieve an instruction due to an Icache miss. Rather, it is the part of the Instruction Cache (ICache) miss time where no bytes are available for the decoder.	
9CH	00H	UOPS_NOT_DELIVERED.ANY	This event is used to measure front-end inefficiencies, i.e., when the front end of the machine is not delivering uops to the back end and the back end has not stalled. This event can be used to identify if the machine is truly front-end bound. When this event occurs, it is an indication that the front end of the machine is operating at less than its theoretical peak performance. Background: We can think of the processor pipeline as being divided into 2 broader parts: the front end and the back end. The front end is responsible for fetching the instruction, decoding into uops in machine understandable format and putting them into a uop queue to be consumed by the back end. The back end then takes these uops and allocates the required resources. When all resources are ready, uops are executed. If the back end is not ready to accept uops from the front end, then we do not want to count these as front-end bottlenecks. However, whenever we have bottlenecks in the back end, we will have allocation unit stalls and eventually force the front end to wait until the back end is ready to receive more uops. This event counts only when the back end is requesting more micro-uops and the front end is not able to provide them. When 3 uops are requested and no uops are delivered, the event counts 3. When 3 are requested, and only 1 is delivered, the event counts 2. When only 2 are delivered, the event counts 1. Alternatively stated, the event will not count if 3 uops are delivered, or if the back end is stalled and not requesting any uops at all. Counts indicate missed opportunities for the front end to deliver a uop to the back end. Some examples of conditions that cause front-end efficiencies are: Icache misses, ITLB misses, and decoder restrictions that limit the front-end bandwidth. Known Issues: Some uops require multiple allocation slots. These uops will not be charged as a front end 'not delivered' opportunity, and will be regarded as a back-end problem. For example, the INC instruction has one uop that requires 2 issue slots. A stream of INC instructions will not count as UOPS_NOT_DELIVERED, even though only one instruction can be issued per clock. The low uop issue rate for a stream of INC instructions is considered to be a back-end issue.	
B7H	01H, 02H	OFFCORE_RESPONSE	Requires MSR_OFFCORE_RESP[0,1] to specify request type and response. (Duplicated for both MSRs.)	
C0H	00H	INST_RETIRED.ANY_P	Counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The event continues counting during hardware interrupts, traps, and inside interrupt handlers. This is an architectural performance event. This event uses a programmable general purpose performance counter. *This event is a Precise Event: the EventingRIP field in the PEBS record is precise to the address of the instruction which caused the event. Note: Because PEBS records can be collected only on IA32_PMC0, only one event can use the PEBS facility at a time.	Precise Event

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C2H	00H	UOPS_RETIRED.ANY	Counts uops which have retired.	Precise Event
C2H	01H	UOPS_RETIRED.MS	Counts uops retired that are from the complex flows issued by the micro-sequencer (MS). Counts both the uops from a micro-coded instruction, and the uops that might be generated from a micro-coded assist.	Precise Event
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the processor detects that a program is writing to a code section and has to perform a machine clear because of that modification. Self-modifying code (SMC) causes a severe penalty in all Intel architecture processors.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts machine clears due to memory ordering issues. This occurs when a snoop request happens and the machine is uncertain if memory ordering will be preserved as another core is in the process of modifying the data.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts machine clears due to floating-point (FP) operations needing assists. For instance, if the result was a floating-point denormal, the hardware clears the pipeline and reissues uops to produce the correct IEEE compliant denormal result.	
C3H	08H	MACHINE_CLEARS.DISAMBIGUATION	Counts machine clears due to memory disambiguation. Memory disambiguation happens when a load which has been issued conflicts with a previous un-retired store in the pipeline whose address was not known at issue time, but is later resolved to be the same as the load address.	
C3H	00H	MACHINE_CLEARS.ALL	Counts machine clears for any reason.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts branch instructions retired for all branch types. This is an architectural performance event.	Precise Event
C4H	7EH	BR_INST_RETIRED.JCC	Counts retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was taken and when it was not taken.	Precise Event
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were taken and does not count when the Jcc branch instruction were not taken.	Precise Event
C4H	F9H	BR_INST_RETIRED.CALL	Counts near CALL branch instructions retired.	Precise Event
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts near relative CALL branch instructions retired.	Precise Event
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts near indirect CALL branch instructions retired.	Precise Event
C4H	F7H	BR_INST_RETIRED.RETURN	Counts near return branch instructions retired.	Precise Event
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts near indirect call or near indirect jmp branch instructions retired.	Precise Event
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts far branch instructions retired. This includes far jump, far call and return, and Interrupt call and return.	Precise Event
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts mispredicted branch instructions retired including all branch types.	Precise Event
C5H	7EH	BR_MISP_RETIRED.JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was supposed to be taken and when it was not supposed to be taken (but the processor predicted the opposite condition).	Precise Event

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were supposed to be taken but the processor predicted that it would not be taken.	Precise Event
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts mispredicted near indirect CALL branch instructions retired, where the target address taken was not what the processor predicted.	Precise Event
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts mispredicted near RET branch instructions retired, where the return address taken was not what the processor predicted.	Precise Event
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts mispredicted branch instructions retired that were near indirect call or near indirect jmp, where the target address taken was not what the processor predicted.	Precise Event
CAH	01H	ISSUE_SLOTS_NOT_CONSUMED.RESOURCE_FULL	Counts the number of issue slots per core cycle that were not consumed because of a full resource in the back end. Including but not limited to resources include the Re-order Buffer (ROB), reservation stations (RS), load/store buffers, physical registers, or any other needed machine resource that is currently unavailable. Note that uops must be available for consumption in order for this event to fire. If a uop is not available (Instruction Queue is empty), this event will not count.	
CAH	02H	ISSUE_SLOTS_NOT_CONSUMED.RECOVERY	Counts the number of issue slots per core cycle that were not consumed by the back end because allocation is stalled waiting for a mispredicted jump to retire or other branch-like conditions (e.g. the event is relevant during certain microcode flows). Counts all issue slots blocked while within this window, including slots where uops were not available in the Instruction Queue.	
CAH	00H	ISSUE_SLOTS_NOT_CONSUMED.ANY	Counts the number of issue slots per core cycle that were not consumed by the back end due to either a full resource in the back end (RESOURCE_FULL), or due to the processor recovering from some event (RECOVERY).	
CBH	01H	HW_INTERRUPTS.RECEIVED	Counts hardware interrupts received by the processor.	
CBH	04H	HW_INTERRUPTS.PENDING_AND_MASKED	Counts core cycles during which there are pending interrupts, but interrupts are masked (EFLAGS.IF = 0).	
CDH	00H	CYCLES_DIV_BUSY.ALL	Counts core cycles if either divide unit is busy.	
CDH	01H	CYCLES_DIV_BUSY.IDIV	Counts core cycles if the integer divide unit is busy.	
CDH	02H	CYCLES_DIV_BUSY.FPDIV	Counts core cycles if the floating point divide unit is busy.	
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	Counts the number of load uops retired.	Precise Event
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	Counts the number of store uops retired.	Precise Event
D0H	83H	MEM_UOPS_RETIRED.ALL	Counts the number of memory uops retired that are either a load or a store or both.	Precise Event
D0H	11H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Counts load uops retired that caused a DTLB miss.	Precise Event
D0H	12H	MEM_UOPS_RETIRED.DTLB_MISS_STORES	Counts store uops retired that caused a DTLB miss.	Precise Event

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
D0H	13H	MEM_UOPS_RETIRED.DTLB_MISS	Counts uops retired that had a DTLB miss on load, store or either. Note that when two distinct memory operations to the same page miss the DTLB, only one of them will be recorded as a DTLB miss.	Precise Event
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Counts locked memory uops retired. This includes 'regular' locks and bus locks. To specifically count bus locks only, see the offcore response event. A locked access is one with a lock prefix, or an exchange to memory.	Precise Event
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Counts load uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Counts store uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	43H	MEM_UOPS_RETIRED.SPLIT	Counts memory uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Counts load uops retired that hit the L1 data cache.	Precise Event
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Counts load uops retired that miss the L1 data cache.	Precise Event
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Counts load uops retired that hit in the L2 cache.	Precise Event
0xD1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Counts load uops retired that miss in the L2 cache.	Precise Event
D1H	20H	MEM_LOAD_UOPS_RETIRED.HITM	Counts load uops retired where the cache line containing the data was in the modified state of another core or modules cache (HITM). More specifically, this means that when the load address was checked by other caching agents (typically another processor) in the system, one of those caching agents indicated that they had a dirty copy of the data. Loads that obtain a HITM response incur greater latency than most that is typical for a load. In addition, since HITM indicates that some other processor had this data in its cache, it implies that the data was shared between processors, or potentially was a lock or semaphore value. This event is useful for locating sharing, false sharing, and contended locks.	Precise Event
D1H	40H	MEM_LOAD_UOPS_RETIRED.WCB_HIT	Counts memory load uops retired where the data is retrieved from the WCB (or fill buffer), indicating that the load found its data while that data was in the process of being brought into the L1 cache. Typically a load will receive this indication when some other load or prefetch missed the L1 cache and was in the process of retrieving the cache line containing the data, but that process had not yet finished (and written the data back to the cache). For example, consider load X and Y, both referencing the same cache line that is not in the L1 cache. If load X misses cache first, it obtains and WCB (or fill buffer) begins the process of requesting the data. When load Y requests the data, it will either hit the WCB, or the L1 cache, depending on exactly what time the request to Y occurs.	Precise Event
D1H	80H	MEM_LOAD_UOPS_RETIRED.DRAM_HIT	Counts memory load uops retired where the data is retrieved from DRAM. Event is counted at retirement, so the speculative loads are ignored. A memory load can hit (or miss) the L1 cache, hit (or miss) the L2 cache, hit DRAM, hit in the WCB or receive a HITM response.	Precise Event

Table 19-25. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
E6H	01H	BACLEARS.ALL	Counts the number of times a BACLEAR is signaled for any reason, including, but not limited to indirect branch/call, Jcc (Jump on Conditional Code/Jump if Condition is Met) branch, unconditional branch/call, and returns.	
E6H	08H	BACLEARS.RETURN	Counts BACLEARS on return instructions.	
E6H	10H	BACLEARS.COND	Counts BACLEARS on Jcc (Jump on Conditional Code/Jump if Condition is Met) branches.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the Microcode Sequencer (MS) starts a flow of uops from the MSROM. It does not count every time a uop is read from the MSROM. The most common case that this counts is when a micro-coded instruction is encountered by the front end of the machine. Other cases include when an instruction encounters a fault, trap, or microcode assist of any sort that initiates a flow of uops. The event will count MS startups for uops that are speculative, and subsequently cleared by branch mispredict or a machine clear.	
E9H	01H	DECODE_RESTRICTION. PREDECODE_WRONG	Counts the number of times the prediction (from the pre-decode cache) for instruction length is incorrect.	

19.13 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE SILVERMONT MICROARCHITECTURE

Processors based on the Silvermont microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-26. These processors have the CPUID signatures of 06_37H, 06_4AH, 06_4DH, 06_5AH, and 06_5DH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, “Intel® Xeon® processor 5500 Series,” in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

Table 19-26. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
03H	01H	REHABQ.LD_BLOCK_ST_FORWARD	Loads blocked due to store forward restriction.	This event counts the number of retired loads that were prohibited from receiving forwarded data from the store because of address mismatch.
03H	02H	REHABQ.LD_BLOCK_STD_NOTREADY	Loads blocked due to store data not ready.	This event counts the cases where a forward was technically possible, but did not occur because the store data was not available at the right time.
03H	04H	REHABQ.ST_SPLITS	Store uops that split cache line boundary.	This event counts the number of retire stores that experienced cache line boundary splits.
03H	08H	REHABQ.LD_SPLITS	Load uops that split cache line boundary.	This event counts the number of retire loads that experienced cache line boundary splits.
03H	10H	REHABQ.LOCK	Uops with lock semantics.	This event counts the number of retired memory operations with lock semantics. These are either implicit locked instructions such as the XCHG instruction or instructions with an explicit LOCK prefix (F0H).
03H	20H	REHABQ.STA_FULL	Store address buffer full.	This event counts the number of retired stores that are delayed because there is not a store address buffer available.

Table 19-26. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
03H	40H	REHABQ.ANY_LD	Any reissued load uops.	This event counts the number of load uops reissued from Rehabq.
03H	80H	REHABQ.ANY_ST	Any reissued store uops.	This event counts the number of store uops reissued from Rehabq.
04H	01H	MEM_UOPS_RETIRED.L1_MISS_LOADS	Loads retired that missed L1 data cache.	This event counts the number of load ops retired that miss in L1 Data cache. Note that prefetch misses will not be counted.
04H	02H	MEM_UOPS_RETIRED.L2_HIT_LOADS	Loads retired that hit L2.	This event counts the number of load micro-ops retired that hit L2.
04H	04H	MEM_UOPS_RETIRED.L2_MISS_LOADS	Loads retired that missed L2.	This event counts the number of load micro-ops retired that missed L2.
04H	08H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Loads missed DTLB.	This event counts the number of load ops retired that had DTLB miss.
04H	10H	MEM_UOPS_RETIRED.UTLB_MISS	Loads missed UTLB.	This event counts the number of load ops retired that had UTLB miss.
04H	20H	MEM_UOPS_RETIRED.HITM	Cross core or cross module hitm.	This event counts the number of load ops retired that got data from the other core or from the other module.
04H	40H	MEM_UOPS_RETIRED.ALL_LOADS	All Loads.	This event counts the number of load ops retired.
04H	80H	MEM_UOP_RETIRED.ALL_STORES	All Stores.	This event counts the number of store ops retired.
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Duration of D-side page-walks in core cycles.	This event counts every cycle when a D-side (walks due to a load) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Duration of I-side page-walks in core cycles.	This event counts every cycle when an I-side (walks due to an instruction fetch) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.
05H	03H	PAGE_WALKS.WALKS	Total number of page-walks that are completed (I-side and D-side).	This event counts when a data (D) page walk or an instruction (I) page walk is completed or started. Since a page walk implies a TLB miss, the number of TLB misses can be counted by counting the number of pagewalks. Edge trigger bit must be set. Clear Edge to count the number of cycles.
2EH	41H	LONGEST_LAT_CACHE.MISS	L2 cache request misses.	This event counts the total number of L2 cache references and the number of L2 cache misses respectively. L3 is not supported in Silvermont microarchitecture.
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	L2 cache requests from this core.	This event counts requests originating from the core that references a cache line in the L2 cache. L3 is not supported in Silvermont microarchitecture.
30H	00H	L2_REJECT_XQ.ALL	Counts the number of request from the L2 that were not accepted into the XQ.	This event counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the IDI link. The XQ may reject transactions from the L2Q (non-cacheable requests), BBS (L2 misses) and WOB (L2 write-back victims).

Table 19-26. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of request that were not accepted into the L2Q because the L2Q is FULL.	This event counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to insure fairness between cores, or to delay a core's dirty eviction when the address conflicts incoming external snoops. (Note that L2 prefetcher requests that are dropped are not counted by this event.).
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time.
N/A	N/A	CPU_CLK_UNHALTED.CORE	Instructions retired.	This uses the fixed counter 1 to count the same condition as CPU_CLK_UNHALTED.CORE_P does.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Reference cycles when core is not halted.	This event counts the number of reference cycles that the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time. This event is not affected by core frequency changes but counts as if the core is running at the maximum frequency all the time.
N/A	N/A	CPU_CLK_UNHALTED.REF_TSC	Instructions retired.	This uses the fixed counter 2 to count the same condition as CPU_CLK_UNHALTED.REF_P does.
80H	01H	ICACHE.HIT	Instruction fetches from Icache.	This event counts all instruction fetches from the instruction cache.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.6.2.	Requires MSR_OFFCORE_RESP0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.6.2.	Requires MSR_OFFCORE_RESP1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (PEBS supported with IA32_PMC0).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
N/A	N/A	INST_RETIRED.ANY	Instructions retired.	This uses the fixed counter 0 to count the same condition as INST_RETIRED.ANY_P does.
C2H	01H	UOPS_RETIRED.MS	MSROM micro-ops retired.	This event counts the number of micro-ops retired that were supplied from MSROM.
C2H	10H	UOPS_RETIRED.ALL	Micro-ops retired.	This event counts the number of micro-ops retired.
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.

Table 19-26. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Stalls due to Memory ordering.	This event counts the number of times that pipeline was cleared due to memory ordering issues.
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Stalls due to FP assists.	This event counts the number of times that pipeline stalled due to FP operations needing assists.
C3H	08H	MACHINE_CLEARS.ALL	Stalls due to any causes.	This event counts the number of times that pipeline stalled due to due to any causes (including SMC, MO, FP assist, etc.).
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Retired branch instructions.	This event counts the number of branch instructions retired.
C4H	7EH	BR_INST_RETIRED.JCC	Retired branch instructions that were conditional jumps.	This event counts the number of branch instructions retired that were conditional jumps.
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Retired far branch instructions.	This event counts the number of far branch instructions retired.
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Retired instructions of near indirect Jmp or call.	This event counts the number of branch instructions retired that were near indirect call or near indirect jmp.
C4H	F7H	BR_INST_RETIRED.RETURN	Retired near return instructions.	This event counts the number of near RET branch instructions retired.
C4H	F9H	BR_INST_RETIRED.CALL	Retired near call instructions.	This event counts the number of near CALL branch instructions retired.
C4H	FBH	BR_INST_RETIRED.IND_CALL	Retired near indirect call instructions.	This event counts the number of near indirect CALL branch instructions retired.
C4H	FDH	BR_INST_RETIRED.REL_CALL	Retired near relative call instructions.	This event counts the number of near relative CALL branch instructions retired.
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Retired conditional jumps that were taken.	This event counts the number of branch instructions retired that were conditional jumps and taken.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Retired mispredicted branch instructions.	This event counts the number of mispredicted branch instructions retired.
C5H	7EH	BR_MISP_RETIRED.JCC	Retired mispredicted conditional jumps.	This event counts the number of mispredicted branch instructions retired that were conditional jumps.
C5H	BFH	BR_MISP_RETIRED.FAR	Retired mispredicted far branch instructions.	This event counts the number of mispredicted far branch instructions retired.
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Retired mispredicted instructions of near indirect Jmp or call.	This event counts the number of mispredicted branch instructions retired that were near indirect call or near indirect jmp.
C5H	F7H	BR_MISP_RETIRED.RETURN	Retired mispredicted near return instructions.	This event counts the number of mispredicted near RET branch instructions retired.
C5H	F9H	BR_MISP_RETIRED.CALL	Retired mispredicted near call instructions.	This event counts the number of mispredicted near CALL branch instructions retired.
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Retired mispredicted near indirect call instructions.	This event counts the number of mispredicted near indirect CALL branch instructions retired.
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Retired mispredicted near relative call instructions	This event counts the number of mispredicted near relative CALL branch instructions retired.

Table 19-26. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Retired mispredicted conditional jumps that were taken.	This event counts the number of mispredicted branch instructions retired that were conditional jumps and taken.
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of cycles when no uops are allocated and a RATstall is asserted.	Counts the number of cycles when no uops are allocated and a RATstall is asserted.
CAH	3FH	NO_ALLOC_CYCLES.ALL	Front end not delivering.	This event counts the number of cycles when the front end does not provide any instructions to be allocated for any reason.
CAH	50H	NO_ALLOC_CYCLES.NOT_DELIVERED	Front end not delivering back end not stalled.	This event counts the number of cycles when the front end does not provide any instructions to be allocated but the back end is not stalled.
CBH	01H	RS_FULL_STALL.MEC	MEC RS full.	This event counts the number of cycles the allocation pipe line stalled due to the RS for the MEC cluster is full.
CBH	1FH	RS_FULL_STALL.ALL	Any RS full.	This event counts the number of cycles that the allocation pipe line stalled due to any one of the RS is full.
CDH	01H	CYCLES_DIV_BUSY.ANY	Divider Busy.	This event counts the number of cycles the divider is busy.
E6H	01H	BACLEARS.ALL	BACLEARS asserted for any branch.	This event counts the number of baclears for any type of branch.
E6H	08H	BACLEARS.RETURN	BACLEARS asserted for return branch.	This event counts the number of baclears for return branches.
E6H	10H	BACLEARS.COND	BACLEARS asserted for conditional branch.	This event counts the number of baclears for conditional branches.
E7H	01H	MS_DECODED.MS_ENTRY	MS Decode starts.	This event counts the number of times the MSROM starts a flow of UOPS.

19.13.1 Performance Monitoring Events for Processors Based on the Airmont Microarchitecture

Intel processors based on the Airmont microarchitecture support the same architectural and the non-architectural performance monitoring events as processors based on the Silvermont microarchitecture. All of the events listed in Table 19-26 apply. These processors have the CPUID signatures that include 06_4CH.

19.14 PERFORMANCE MONITORING EVENTS FOR 45 NM AND 32 NM INTEL® ATOM™ PROCESSORS

45 nm and 32 nm processors based on the Intel® Atom™ microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter listed in Table 19-23. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-27.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors

Event Num.	Umask Value	Event Name	Definition	Description and Comment
02H	81H	STORe_FORWARDS.GOOD	Good store forwards.	This event counts the number of times store data was forwarded directly to a load.
06H	00H	SEGMENT_REG_LOADS.ANY	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.
07H	01H	PREFETCH.PREFETCHT0	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	06H	PREFETCH.SW_L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.
07H	08H	PREFETCH.PREFETCHNTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
08H	07H	DATA_TLB_MISSES.DTLB_MISS	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	05H	DATA_TLB_MISSES.DTLB_MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	09H	DATA_TLB_MISSES.L0_DTLB_MISS_LD	L0_DTLB misses due to load operations.	This event counts the number of L0_DTLB misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	06H	DATA_TLB_MISSES.DTLB_MISS_ST	DTLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
0CH	03H	PAGE_WALKS.WALKS	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. This can hint to whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be set.
0CH	03H	PAGE_WALKS.CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. This can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be cleared.
10H	01H	X87_COMP_OPS_EXE.ANY.S	Floating point computational micro-ops executed.	This event counts the number of x87 floating point computational micro-ops executed.
10H	81H	X87_COMP_OPS_EXE.ANY.AR	Floating point computational micro-ops retired.	This event counts the number of x87 floating point computational micro-ops retired.
11H	01H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.
11H	81H	FP_ASSIST.AR	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.
12H	01H	MUL.S	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations.
12H	81H	MUL.AR	Multiply operations retired.	This event counts the number of multiply operations retired. This includes integer as well as floating point multiply operations.
13H	01H	DIV.S	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed.
13H	81H	DIV.AR	Divide operations retired.	This event counts the number of divide operations retired. This includes integer divides, floating point divides and square-root operations executed.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
14H	01H	CYCLES_DIV_BUSY	Cycles the driver is busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE.
21H	See Table 18-3	L2_ADS	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. This event can count occurrences for this core or both cores.
22H	See Table 18-3	L2_DBUS_BUSY	Cycles the L2 cache data bus is busy.	This event counts core cycles during which the L2 cache data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. The count will increment by two for a full cache-line request.
24H	See Table 18-3 and Table 18-5	L2_LINES_IN	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. This event can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-3	L2_M_LINES_IN	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.
26H	See Table 18-3 and Table 18-5	L2_LINES_OUT	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-3 and Table 18-5	L2_M_LINES_OUT	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a shared-state in one of the L1 data caches. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
28H	See Table 18-3 and Table 18-6	L2_IFETCH	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the ICache. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
29H	See Table 18-3, Table 18-5 and Table 18-6	L2_LD	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. This event can count occurrences for this core or both cores. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together or separately. - of accesses to cache lines at different MESI states.
2AH	See Table 18-3 and Table 18-6	L2_ST	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2BH	See Table 18-3 and Table 18-6	L2_LOCK	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2EH	See Table 18-3, Table 18-5 and Table 18-6	L2_RQSTS	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together, or separately. - of accesses to cache lines at different MESI states.
2EH	41H	L2_RQSTS.SELF.DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF.DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
30H	See Table 18-3, Table 18-5 and Table 18-6	L2_REJECT_BUSQ	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: - The bus queue is full. - The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. - For this core or both cores. - Due to demand requests and L2 hardware prefetch requests together, or separately. - Of accesses to cache lines at different MESI states.
32H	See Table 18-3	L2_NO_REQ	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests are pending.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions.	This event counts the number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions that include a frequency change, either with or without VID change. This event is incremented only while the counting core is in C0 state. In situations where an EIST transition was caused by hardware as a result of CxE state transitions, those EIST transitions will also be registered in this event. Enhanced Intel Speedstep Technology transitions are commonly initiated by OS, but can be initiated by HW internally. For example: CxE states are C-states (C1,C2,C3…) which not only place the CPU into a sleep state by turning off the clock and other components, but also lower the voltage (which reduces the leakage power consumption). The same is true for thermal throttling transition which uses Enhanced Intel Speedstep Technology internally.
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond, and returns to normal when the temperature falls below the thermal trip threshold temperature.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time. In systems with a constant core frequency, this event can give you a measurement of the elapsed time while the core was not in halt state by dividing the event count by the core frequency. -This is an architectural performance event. - The event CPU_CLK_UNHALTED.CORE_P is counted by a programmable counter. - The event CPU_CLK_UNHALTED.CORE is counted by a designated fixed counter, leaving the two programmable counters available for other events.
3CH	01H	CPU_CLK_UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give you a measurement of the elapsed time while the core was not in the halt state, by dividing the event count by the bus frequency. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_UNHALTED.NO_OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted, and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	21H	L1D_CACHE.LD	L1 Cacheable Data Reads.	This event counts the number of data reads from cacheable memory.
40H	22H	L1D_CACHE.ST	L1 Cacheable Data Writes.	This event counts the number of data writes to cacheable memory.
60H	See Table 18-3 and Table 18-4.	BUS_REQUEST_OUTSTANDING	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
61H	See Table 18-4.	BUS_BNR_DRV	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
62H	See Table 18-4.	BUS_DRDY_CLOCKS	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. This event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
63H	See Table 18-3 and Table 18-4.	BUS_LOCK_CLOCKS	Bus cycles when a LOCK signal is asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: - Uncacheable memory. - Locked operation that spans two cache lines. - Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
64H	See Table 18-3.	BUS_DATA_RCV	Bus cycles while processor receives data.	This event counts the number of cycles during which the processor is busy receiving data. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
65H	See Table 18-3 and Table 18-4.	BUS_TRANS_BRD	Burst read bus transactions.	This event counts the number of burst read transactions including: - L1 data cache read misses (and L1 data cache hardware prefetches). - L2 hardware prefetches by the DPL and L2 streamer. - IFU read misses of cacheable lines. It does not include RFO transactions.
66H	See Table 18-3 and Table 18-4.	BUS_TRANS_RFO	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. This event also counts RFO bus transactions due to locked operations.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
67H	See Table 18-3 and Table 18-4.	BUS_TRANS_WB	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-3 and Table 18-4.	BUS_TRANS_IFETCH	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-3 and Table 18-4.	BUS_TRANS_INVAL	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: - A store operation hits a shared line in the L2 cache. - A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-3 and Table 18-4.	BUS_TRANS_PWR	Partial write bus transaction.	This event counts partial write bus transactions.
6BH	See Table 18-3 and Table 18-4.	BUS_TRANS_P	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-3 and Table 18-4.	BUS_TRANS_IO	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-3 and Table 18-4.	BUS_TRANS_DEF	Deferred bus transactions.	This event counts the number of deferred transactions.
6EH	See Table 18-3 and Table 18-4.	BUS_TRANS_BURST	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: - Burst reads. - RFOs. - Explicit writebacks. - Write combine lines.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
6FH	See Table 18-3 and Table 18-4.	BUS_TRANS_MEM	Memory bus transactions.	This event counts all memory bus transactions including: - Burst transactions. - Partial reads and writes. - Invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_INVAL.
70H	See Table 18-3 and Table 18-4.	BUS_TRANS_ANY	All bus transactions.	This event counts all bus transactions. This includes: - Memory transactions. - IO transactions (non memory-mapped). - Deferred transaction completion. - Other less frequent transactions, such as interrupts.
77H	See Table 18-3 and Table 18-6.	EXT_SNOOP	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7AH	See Table 18-4.	BUS_HIT_DRV	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7BH	See Table 18-4.	BUS_HITM_DRV	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7DH	See Table 18-3.	BUSQ_EMPTY	Bus queue is empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7EH	See Table 18-3 and Table 18-4.	SNOOP_STALL_DRV	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. During the snoop stall cycles no new bus transactions requiring a snoop response can be initiated on the bus. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7FH	See Table 18-3.	BUS_IO_WAIT	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. This event counts IO requests from the core.
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	04H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes.
82H	02H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches that miss the ITLB.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
AAH	02H	MACRO_INSTS.CISC_DECODED	CISC macro instructions decoded.	This event counts the number of complex instructions decoded, but not necessarily executed or retired. Only one complex instruction can be decoded at a time.
AAH	03H	MACRO_INSTS.ALL_DECODED	All Instructions decoded.	This event counts the number of instructions decoded.
B0H	00H	SIMD_UOPS_EXEC.S	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. This event does not count MOVQ and MOVD stores from register to memory.
B0H	80H	SIMD_UOPS_EXEC.AR	SIMD micro-ops retired (excluding stores).	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	00H	SIMD_SAT_UOP_EXEC.S	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	80H	SIMD_SAT_UOP_EXEC.AR	SIMD saturated arithmetic micro-ops retired.	This event counts the number of SIMD saturated arithmetic micro-ops retired.
B3H	01H	SIMD_UOP_TYPE_EXEC.MUL.S	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	81H	SIMD_UOP_TYPE_EXEC.MUL.AR	SIMD packed multiply micro-ops retired.	This event counts the number of SIMD packed multiply micro-ops retired.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHIFT.S	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	82H	SIMD_UOP_TYPE_EXEC.SHIFT.AR	SIMD packed shift micro-ops retired.	This event counts the number of SIMD packed shift micro-ops retired.
B3H	04H	SIMD_UOP_TYPE_EXEC.PACK.S	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	84H	SIMD_UOP_TYPE_EXEC.PACK.AR	SIMD pack micro-ops retired.	This event counts the number of SIMD pack micro-ops retired.
B3H	08H	SIMD_UOP_TYPE_EXEC.UNPACK.S	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	88H	SIMD_UOP_TYPE_EXEC.UNPACK.AR	SIMD unpack micro-ops retired.	This event counts the number of SIMD unpack micro-ops retired.
B3H	10H	SIMD_UOP_TYPE_EXEC.LOGICAL.S	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.
B3H	90H	SIMD_UOP_TYPE_EXEC.LOGICAL.AR	SIMD packed logical micro-ops retired.	This event counts the number of SIMD packed logical micro-ops retired.
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.S	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
B3H	A0H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.AR	SIMD packed arithmetic micro-ops retired.	This event counts the number of SIMD packed arithmetic micro-ops retired.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (precise event).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
N/A	00H	INST_RETIRED.ANY	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
C2H	10H	UOPS_RETIRED.ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. **This is an architectural performance event.**
C4H	01H	BR_INST_RETIRED.PRED_NOT_TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRED_NOT_TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0AH	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
				To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C4H	0FH	BR_INST_RETIRED.ANY1	Retired branch instructions.	This event counts the number of branch instructions retired that were mispredicted. This event is a duplicate of BR_INST_RETIRED.MISPRED.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature. To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C6H	01H	CYCLES_INT_MASKED.CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_MASKED.CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.
C7H	10H	SIMD_INST_RETIRED.VECTOR	Retired Streaming SIMD Extensions 2 (SSE2) vector instructions.	This event counts the number of SSE2 vector instructions retired.
C7H	1FH	SIMD_INST_RETIRED.ANY	Retired Streaming SIMD instructions.	This event counts the overall number of SIMD instructions retired. To count each type of SIMD instruction separately, use the following events: SIMD_INST_RETIRED.PACKED_SINGLE SIMD_INST_RETIRED.SCALAR_SINGLE SIMD_INST_RETIRED.PACKED_DOUBLE SIMD_INST_RETIRED.SCALAR_DOUBLE SIMD_INST_RETIRED.VECTOR.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor. This event will count twice for dual-pipe micro-ops.
CAH	01H	SIMD_COMP_INST_RETIRED.PACKED_SINGLE	Retired computational Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_INST_RETIRED.SCALAR_SINGLE	Retired computational Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_INST_RETIRED.PACKED_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.

Table 19-27. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
CAH	08H	SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CBH	01H	MEM_LOAD_RETIRED.L2_HIT	Retired loads that hit the L2 cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache and hit the L2 cache.
CBH	02H	MEM_LOAD_RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache.
CBH	04H	MEM_LOAD_RETIRED.DTLB_MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault.
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed after MMX™ technology code has changed the MMX state in the floating point stack. For example, these assists are required in the following cases. Streaming SIMD Extensions (SSE) instructions: 1. Denormal input when the DAZ (Denormals Are Zeros) flag is off. 2. Underflow result when the FTZ (Flush To Zero) flag is off.
CEH	00H	SIMD_INSTR_RETIRED	SIMD Instructions retired.	This event counts the number of SIMD instructions that retired.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
E0H	01H	BR_INST_DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	01H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event and the BTB is flushed. This occurs mainly after task switches.
E6H	01H	BACLEARS.ANY	BACLEARS asserted.	This event counts the number of times the front end is redirected for a branch prediction, mainly when an early branch prediction is corrected by other branch handling mechanisms in the front end. This can occur if the code has many branches such that they cannot be consumed by the branch predictor. Each Baclear asserted costs approximately 7 cycles. The effect on total execution time depends on the surrounding code.

19.15 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Table 19-28 lists non-architectural performance events for Intel® Core™ Duo processors. If a non-architectural event requires qualification in core specificity, it is indicated in the comment column. Table 19-28 also applies to Intel® Core™ Solo processors; bits in the unit mask corresponding to core-specificity are reserved and should be 00B.

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
03H	LD_Blocks	00H	Load operations delayed due to store buffer blocks. The preceding store may be blocked due to unknown address, unknown data, or conflict due to partial overlap between the load and store.	
04H	SD_Drains	00H	Cycles while draining store buffers.	
05H	Misalign_Mem_Ref	00H	Misaligned data memory references (MOB splits of loads and stores).	
06H	Seg_Reg_Loads	00H	Segment register loads.	
07H	SSE_PrefNta_Ret	00H	SSE software prefetch instruction PREFETCHNTA retired.	
07H	SSE_PrefT1_Ret	01H	SSE software prefetch instruction PREFETCHT1 retired.	
07H	SSE_PrefT2_Ret	02H	SSE software prefetch instruction PREFETCHT2 retired.	
07H	SSE_NTStores_Ret	03H	SSE streaming store instruction retired.	
10H	FP_Comps_Op_Exe	00H	FP computational Instruction executed. FADD, FSUB, FCOM, FMULs, MUL, IMUL, FDIVs, DIV, IDIV, FPREMs, FSQRT are included; but exclude FADD or FMUL used in the middle of a transcendental instruction.	
11H	FP_Assist	00H	FP exceptions experienced microcode assists.	IA32_PMC1 only.
12H	Mul	00H	Multiply operations (a speculative count, including FP and integer multiplies).	IA32_PMC1 only.
13H	Div	00H	Divide operations (a speculative count, including FP and integer divisions).	IA32_PMC1 only.
14H	Cycles_Div_Busy	00H	Cycles the divider is busy.	IA32_PMC0 only.
21H	L2_ADS	00H	L2 Address strobes.	Requires core-specificity.
22H	Dbus_Busy	00H	Core cycle during which data bus was busy (increments by 4).	Requires core-specificity.
23H	Dbus_Busy_Rd	00H	Cycles data bus is busy transferring data to a core (increments by 4).	Requires core-specificity.
24H	L2_Lines_In	00H	L2 cache lines allocated.	Requires core-specificity and HW prefetch qualification.
25H	L2_M_Lines_In	00H	L2 Modified-state cache lines allocated.	Requires core-specificity.

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
26H	L2_Lines_Out	00H	L2 cache lines evicted.	Requires core-specificity and HW prefetch qualification.
27H	L2_M_Lines_Out	00H	L2 Modified-state cache lines evicted.	
28H	L2_IFetch	Requires MESI qualification	L2 instruction fetches from instruction fetch unit (includes speculative fetches).	Requires core-specificity.
29H	L2_LD	Requires MESI qualification	L2 cache reads.	Requires core-specificity.
2AH	L2_ST	Requires MESI qualification	L2 cache writes (includes speculation).	Requires core-specificity.
2EH	L2_Rqsts	Requires MESI qualification	L2 cache reference requests.	Requires core-specificity, HW prefetch qualification.
30H	L2_Reject_Cycles	Requires MESI qualification	Cycles L2 is busy and rejecting new requests.	
32H	L2_No_Request_ Cycles	Requires MESI qualification	Cycles there is no request to access L2.	
3AH	EST_Trans_All	00H	Any Intel Enhanced SpeedStep(R) Technology transitions.	
3AH	EST_Trans_All	10H	Intel Enhanced SpeedStep Technology frequency transitions.	
3BH	Thermal_Trip	C0H	Duration in a thermal trip based on the current core clock.	Use edge trigger to count occurrence.
3CH	NonHlt_Ref_Cycles	01H	Non-halted bus cycles.	
3CH	Serial_Execution_ Cycles	02H	Non-halted bus cycles of this core executing code while the other core is halted.	
40H	DCache_Cache_LD	Requires MESI qualification	L1 cacheable data read operations.	
41H	DCache_Cache_ST	Requires MESI qualification	L1 cacheable data write operations.	
42H	DCache_Cache_ Lock	Requires MESI qualification	L1 cacheable lock read operations to invalid state.	
43H	Data_Mem_Ref	01H	L1 data read and writes of cacheable and non-cacheable types.	
44H	Data_Mem_Cache_ Ref	02H	L1 data cacheable read and write operations.	
45H	DCache_Repl	0FH	L1 data cache line replacements.	
46H	DCache_M_Repl	00H	L1 data M-state cache line allocated.	
47H	DCache_M_Evict	00H	L1 data M-state cache line evicted.	
48H	DCache_Pend_Miss	00H	Weighted cycles of L1 miss outstanding.	Use Cmask =1 to count duration.
49H	Dtlb_Miss	00H	Data references that missed TLB.	
4BH	SSE_PrefNta_Miss	00H	PREFETCHNTA missed all caches.	
4BH	SSE_PrefT1_Miss	01H	PREFETCHT1 missed all caches.	
4BH	SSE_PrefT2_Miss	02H	PREFETCHT2 missed all caches.	
4BH	SSE_NTStores_ Miss	03H	SSE streaming store instruction missed all caches.	

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
4FH	L1_Pref_Req	00H	L1 prefetch requests due to DCU cache misses.	May overcount if request re-submitted.
60H	Bus_Req_Outstanding	00; Requires core-specificity, and agent specificity	Weighted cycles of cacheable bus data read requests. This event counts full-line read request from DCU or HW prefetcher, but not RFO, write, instruction fetches, or others.	Use Cmask =1 to count duration. Use Umask bit 12 to include HWP or exclude HWP separately.
61H	Bus_BNR_Clocks	00H	External bus cycles while BNR asserted.	
62H	Bus_DRDY_Clocks	00H	External bus cycles while DRDY asserted.	Requires agent specificity.
63H	Bus_Locks_Clocks	00H	External bus cycles while bus lock signal asserted.	Requires core specificity.
64H	Bus_Data_Rcv	40H	Number of data chunks received by this processor.	
65H	Bus_Trans_Brd	See comment.	Burst read bus transactions (data or code).	Requires core specificity.
66H	Bus_Trans_RFO	See comment.	Completed read for ownership (RFO) transactions.	Requires agent specificity. Requires core specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
68H	Bus_Trans_Ifetch	See comment.	Completed instruction fetch transactions.	
69H	Bus_Trans_Inval	See comment.	Completed invalidate transactions.	
6AH	Bus_Trans_Pwr	See comment.	Completed partial write transactions.	
6BH	Bus_Trans_P	See comment.	Completed partial transactions (include partial read + partial write + line write).	
6CH	Bus_Trans_IO	See comment.	Completed I/O transactions (read and write).	
6DH	Bus_Trans_Def	20H	Completed defer transactions.	Requires core specificity. Retried transaction may be counted more than once.
67H	Bus_Trans_WB	C0H	Completed writeback transactions from DCU (does not include L2 writebacks).	Requires agent specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
6EH	Bus_Trans_Burst	C0H	Completed burst transactions (full line transactions include reads, write, RFO, and writebacks).	
6FH	Bus_Trans_Mem	C0H	Completed memory transactions. This includes Bus_Trans_Burst + Bus_Trans_P+Bus_Trans_Inval.	
70H	Bus_Trans_Any	C0H	Any completed bus transactions.	
77H	Bus_Snoops	00H	Counts any snoop on the bus.	Requires MESI qualification. Requires agent specificity.
78H	DCU_Snoop_To_Share	01H	DCU snoops to share-state L1 cache line due to L1 misses.	Requires core specificity.
7DH	Bus_Not_In_Use	00H	Number of cycles there is no transaction from the core.	Requires core specificity.
7EH	Bus_Snoop_Stall	00H	Number of bus cycles while bus snoop is stalled.	

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
80H	ICache_Reads	00H	Number of instruction fetches from ICache, streaming buffers (both cacheable and uncacheable fetches).	
81H	ICache_Misses	00H	Number of instruction fetch misses from ICache, streaming buffers.	
85H	ITLB_Misses	00H	Number of iITLB misses.	
86H	IFU_Mem_Stall	00H	Cycles IFU is stalled while waiting for data from memory.	
87H	ILD_Stall	00H	Number of instruction length decoder stalls (Counts number of LCP stalls).	
88H	Br_Inst_Exec	00H	Branch instruction executed (includes speculation).	
89H	Br_Missp_Exec	00H	Branch instructions executed and mispredicted at execution (includes branches that do not have prediction or mispredicted).	
8AH	Br_BAC_Missp_Exec	00H	Branch instructions executed that were mispredicted at front end.	
8BH	Br_Cnd_Exec	00H	Conditional branch instructions executed.	
8CH	Br_Cnd_Missp_Exec	00H	Conditional branch instructions executed that were mispredicted.	
8DH	Br_Ind_Exec	00H	Indirect branch instructions executed.	
8EH	Br_Ind_Missp_Exec	00H	Indirect branch instructions executed that were mispredicted.	
8FH	Br_Ret_Exec	00H	Return branch instructions executed.	
90H	Br_Ret_Missp_Exec	00H	Return branch instructions executed that were mispredicted.	
91H	Br_Ret_BAC_Missp_Exec	00H	Return branch instructions executed that were mispredicted at the front end.	
92H	Br_Call_Exec	00H	Return call instructions executed.	
93H	Br_Call_Missp_Exec	00H	Return call instructions executed that were mispredicted.	
94H	Br_Ind_Call_Exec	00H	Indirect call branch instructions executed.	
A2H	Resource_Stall	00H	Cycles while there is a resource related stall (renaming, buffer entries) as seen by allocator.	
B0H	MMX_Instr_Exec	00H	Number of MMX instructions executed (does not include MOVQ and MOVD stores).	
B1H	SIMD_Int_Sat_Exec	00H	Number of SIMD Integer saturating instructions executed.	
B3H	SIMD_Int_Pmul_Exec	01H	Number of SIMD Integer packed multiply instructions executed.	
B3H	SIMD_Int_Psft_Exec	02H	Number of SIMD Integer packed shift instructions executed.	
B3H	SIMD_Int_Pck_Exec	04H	Number of SIMD Integer pack operations instruction executed.	
B3H	SIMD_Int_Upck_Exec	08H	Number of SIMD Integer unpack instructions executed.	

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
B3H	SIMD_Int_Plog_Exec	10H	Number of SIMD Integer packed logical instructions executed.	
B3H	SIMD_Int_Pari_Exec	20H	Number of SIMD Integer packed arithmetic instructions executed.	
C0H	Instr_Ret	00H	Number of instruction retired (Macro fused instruction count as 2).	
C1H	FP_Comp_Instr_Ret	00H	Number of FP compute instructions retired (X87 instruction or instruction that contains X87 operations).	Use IA32_PMC0 only.
C2H	Uops_Ret	00H	Number of micro-ops retired (include fused uops).	
C3H	SMC_Detected	00H	Number of times self-modifying code condition detected.	
C4H	Br_Instr_Ret	00H	Number of branch instructions retired.	
C5H	Br_MisPred_Ret	00H	Number of mispredicted branch instructions retired.	
C6H	Cycles_Int_Masked	00H	Cycles while interrupt is disabled.	
C7H	Cycles_Int_Pedning_Masked	00H	Cycles while interrupt is disabled and interrupts are pending.	
C8H	HW_Int_Rx	00H	Number of hardware interrupts received.	
C9H	Br_Taken_Ret	00H	Number of taken branch instruction retired.	
CAH	Br_MisPred_Taken_Ret	00H	Number of taken and mispredicted branch instructions retired.	
CCH	MMX_FP_Trans	00H	Number of transitions from MMX to X87.	
CCH	FP_MMX_Trans	01H	Number of transitions from X87 to MMX.	
CDH	MMX_Assist	00H	Number of EMMS executed.	
CEH	MMX_Instr_Ret	00H	Number of MMX instruction retired.	
D0H	Instr_Decoded	00H	Number of instruction decoded.	
D7H	ESP_Uops	00H	Number of ESP folding instruction decoded.	
D8H	SIMD_FP_SP_Ret	00H	Number of SSE/SSE2 single precision instructions retired (packed and scalar).	
D8H	SIMD_FP_SP_S_Ret	01H	Number of SSE/SSE2 scalar single precision instructions retired.	
D8H	SIMD_FP_DP_P_Ret	02H	Number of SSE/SSE2 packed double precision instructions retired.	
D8H	SIMD_FP_DP_S_Ret	03H	Number of SSE/SSE2 scalar double precision instructions retired.	
D8H	SIMD_Int_128_Ret	04H	Number of SSE2 128 bit integer instructions retired.	
D9H	SIMD_FP_SP_P_Comp_Ret	00H	Number of SSE/SSE2 packed single precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_SP_S_Comp_Ret	01H	Number of SSE/SSE2 scalar single precision compute instructions retired (does not include AND, OR, XOR).	

Table 19-28. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
D9H	SIMD_FP_DP_P_ Comp_Ret	02H	Number of SSE/SSE2 packed double precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_DP_S_ Comp_Ret	03H	Number of SSE/SSE2 scalar double precision compute instructions retired (does not include AND, OR, XOR).	
DAH	Fused_Uops_Ret	00H	All fused uops retired.	
DAH	Fused_Ld_Uops_ Ret	01H	Fused load uops retired.	
DAH	Fused_St_Uops_Ret	02H	Fused store uops retired.	
DBH	Unfusion	00H	Number of unfusion events in the ROB (due to exception).	
E0H	Br_Instr_Decoded	00H	Branch instructions decoded.	
E2H	BTB_Misses	00H	Number of branches the BTB did not produce a prediction.	
E4H	Br_Bogus	00H	Number of bogus branches.	
E6H	BAClears	00H	Number of BAClears asserted.	
F0H	Pref_Rqsts_Up	00H	Number of hardware prefetch requests issued in forward streams.	
F8H	Pref_Rqsts_Dn	00H	Number of hardware prefetch requests issued in backward streams.	

19.16 PENTIUM® 4 AND INTEL® XEON® PROCESSOR PERFORMANCE-MONITORING EVENTS

Tables 19-29, 19-30 and 19-31 list performance-monitoring events that can be counted or sampled on processors based on Intel NetBurst® microarchitecture. Table 19-29 lists the non-retirement events, and Table 19-30 lists the at-retirement events. Tables 19-32, 19-33, and 19-34 describes three sets of parameters that are available for three of the at-retirement counting events defined in Table 19-30. Table 19-35 shows which of the non-retirement and at retirement events are logical processor specific (TS) (see Section 18.16.4, "Performance Monitoring Events") and which are non-logical processor specific (TI).

Some of the Pentium 4 and Intel Xeon processor performance-monitoring events may be available only to specific models. The performance-monitoring events listed in Tables 19-29 and 19-30 apply to processors with CPUID signature that matches family encoding 15, model encoding 0, 1, 2 3, 4, or 6. Table applies to processors with a CPUID signature that matches family encoding 15, model encoding 3, 4 or 6.

The functionality of performance-monitoring events in Pentium 4 and Intel Xeon processors is also available when IA-32e mode is enabled.

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
TC_deliver_mode			This event counts the duration (in clock cycles) of the operating modes of the trace cache and decode engine in the processor package. The mode is specified by one or more of the event mask bits.

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit	
		0: DD	Both logical processors are in deliver mode.
		1: DB	Logical processor 0 is in deliver mode and logical processor 1 is in build mode.
		2: DI	Logical processor 0 is in deliver mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow.
		3: BD	Logical processor 0 is in build mode and logical processor 1 is in deliver mode.
		4: BB	Both logical processors are in build mode.
		5: BI	Logical processor 0 is in build mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow.
		6: ID	Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in deliver mode.
		7: IB	Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in build mode.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If only one logical processor is available from a physical processor package, the event mask should be interpreted as logical processor 1 is halted. Event mask bit 2 was previously known as “DELIVER”, bit 5 was previously known as “BUILD”.
BPU_fetch_ request			This event counts instruction fetch requests of specified request type by the Branch Prediction unit. Specify one or more mask bits to qualify the request type(s).
	ESCR restrictions	MSR_BPU_ESCR0 MSR_BPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: TCMISS	ESCR[24:9] Trace cache lookup miss
	CCCR Select	00H	CCCR[15:13]
ITLB_reference			This event counts translations using the Instruction Translation Look-aside Buffer (ITLB).
	ESCR restrictions	MSR_ITLB_ESCR0 MSR_ITLB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	18H	ESCR[31:25]
	ESCR Event Mask	Bit 0: HIT 1: MISS 2: HIT_UC	ESCR[24:9] ITLB hit ITLB miss Uncacheable ITLB hit
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		All page references regardless of the page size are looked up as actual 4-KByte pages. Use the page_walk_type event with the ITMISS mask for a more conservative count.
memory_cancel			This event counts the canceling of various type of request in the Data cache Address Control unit (DAC). Specify one or more mask bits to select the type of requests that are canceled.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 2: ST_RB_FULL 3: 64K_CONF	ESCR[24:9] Replayed because no store request buffer is available. Conflicts due to 64-KByte aliasing.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		All_CACHE_MISS includes uncacheable memory in count.
memory_complete			This event counts the completion of a load split, store split, uncacheable (UC) split, or UC load. Specify one or more mask bits to select the operations to be counted.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]
	ESCR Event Mask	Bit 0: LSC 1: SSC	ESCR[24:9] Load split completed, excluding UC/WC loads. Any split stores completed.
	CCCR Select	02H	CCCR[15:13]
load_port_replay			This event counts replayed events at the load port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_LD	ESCR[24:9] Split load.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
store_port_replay			This event counts replayed events at the store port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_ST	ESCR[24:9] Split store
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
MOB_load_replay			This event triggers if the memory order buffer (MOB) caused a load operation to be replayed. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_MOB_ESCR0 MSR_MOB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 1: NO_STA 3: NO_STD	ESCR[24:9] Replayed because of unknown store address. Replayed because of unknown store data.
		4: PARTIAL_DATA 5: UNALGN_ADDR	Replayed because of partially overlapped data access between the load and store operations. Replayed because the lower 4 bits of the linear address do not match between the load and store operations.
	CCCR Select	02H	CCCR[15:13]
page_walk_type			This event counts various types of page walks that the page miss handler (PMH) performs.
	ESCR restrictions	MSR_PMH_ESCR0 MSR_PMH_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	Bit 0: DTMISS 1: ITMISS	ESCR[24:9] Page walk for a data TLB miss (either load or store). Page walk for an instruction TLB miss.
	CCCR Select	04H	CCCR[15:13]
BSQ_cache_reference			This event counts cache references (2nd level cache or 3rd level cache) as seen by the bus unit. Specify one or more mask bit to select an access according to the access type (read type includes both load and RFO, write type includes writebacks and evictions) and the access result (hit, misses).
	ESCR restrictions	MSR_BSU_ESCR0 MSR_BSU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	0CH	ESCR[31:25]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		Bit 0: RD_2ndL_HITS 1: RD_2ndL_HITE 2: RD_2ndL_HITM 3: RD_3rdL_HITS 4: RD_3rdL_HITE 5: RD_3rdL_HITM	ESCR[24:9] Read 2nd level cache hit Shared (includes load and RFO). Read 2nd level cache hit Exclusive (includes load and RFO). Read 2nd level cache hit Modified (includes load and RFO). Read 3rd level cache hit Shared (includes load and RFO). Read 3rd level cache hit Exclusive (includes load and RFO). Read 3rd level cache hit Modified (includes load and RFO).
	ESCR Event Mask	8: RD_2ndL_MISS 9: RD_3rdL_MISS 10: WR_2ndL_MISS	Read 2nd level cache miss (includes load and RFO). Read 3rd level cache miss (includes load and RFO). A Writeback lookup from DAC misses the 2nd level cache (unlikely to happen).
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: The implementation of this event in current Pentium 4 and Xeon processors treats either a load operation or a request for ownership (RFO) request as a “read” type operation. 2: Currently this event causes both over and undercounting by as much as a factor of two due to an erratum. 3: It is possible for a transaction that is started as a prefetch to change the transaction’s internal status, making it no longer a prefetch. or change the access result status (hit, miss) as seen by this event.
IOQ_allocation			This event counts the various types of transactions on the bus. A count is generated each time a transaction is allocated into the IOQ that matches the specified mask bits. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. Requests are counted once per retry. The event mask bits constitute 4 bit fields. A transaction type is specified by interpreting the values of each bit field. Specify one or more event mask bits in a bit field to select the value of the bit field. Each field (bits 0-4 are one field) are independent of and can be ORed with the others. The request type field is further combined with bit 5 and 6 to form a binary expression. Bits 7 and 8 form a bit field to specify the memory type of the target address. Bits 13 and 14 form a bit field to specify the source agent of the request. Bit 15 affects read operation only. The event is triggered by evaluating the logical expression: (((Request type) OR Bit 5 OR Bit 6) OR (Memory type)) AND (Source agent).

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR restrictions	MSR_FSB_ESCR0, MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1; ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bits	
		0-4 (single field)	Bus request type (use 00001 for invalid or default).
		5: ALL_READ	Count read entries.
		6: ALL_WRITE	Count write entries.
		7: MEM_UC	Count UC memory access entries.
		8: MEM_WC	Count WC memory access entries.
		9: MEM_WT	Count write-through (WT) memory access entries.
		10: MEM_WP	Count write-protected (WP) memory access entries.
		11: MEM_WB	Count WB memory access entries.
		13: OWN	Count all store requests driven by processor, as opposed to other processor or DMA.
		14: OTHER	Count all requests driven by other processors or DMA.
		15: PREFETCH	Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: If PREFETCH bit is cleared, sectors fetched using prefetch are excluded in the counts. If PREFETCH bit is set, all sectors or chunks read are counted. 2: Specify the edge trigger in CCCR to avoid double counting.
			3: The mapping of interpreted bit field values to transaction types may differ with different processor model implementations of the Pentium 4 processor family. Applications that program performance monitoring events should use CPUID to determine processor models when using this event. The logic equations that trigger the event are model-specific (see 4a and 4b below). 4a:For Pentium 4 and Xeon Processors starting with CPUID Model field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)). 4b:For Pentium 4 and Xeon Processors with CPUID Model field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Note that event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5: This event is known to ignore CPL in early implementations of Pentium 4 and Xeon Processors. Both user requests and OS requests are included in the count. This behavior is fixed starting with Pentium 4 and Xeon Processors with CPUID signature F27H (Family 15, Model 2, Stepping 7).

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			6: For write-through (WT) and write-protected (WP) memory types, this event counts reads as the number of 64-byte sectors. Writes are counted by individual chunks. 7: For uncacheable (UC) memory types, this event counts the number of 8-byte chunks allocated. 8: For Pentium 4 and Xeon Processors with CPUID Signature less than F27H, only MSR_FSB_ESCR0 is available.
IOQ_active_ entries			This event counts the number of entries (clipped at 15) in the IOQ that are active. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. The event must be programmed in conjunction with IOQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	01AH	ESCR[30:25]
	ESCR Event Mask		ESCR[24:9]
		Bits	
		0-4 (single field)	Bus request type (use 00001 for invalid or default).
		5: ALL_READ	Count read entries.
		6: ALL_WRITE	Count write entries.
		7: MEM_UC	Count UC memory access entries.
		8: MEM_WC	Count WC memory access entries.
		9: MEM_WT	Count write-through (WT) memory access entries.
		10: MEM_WP	Count write-protected (WP) memory access entries.
		11: MEM_WB	Count WB memory access entries.
		13: OWN	Count all store requests driven by processor, as opposed to other processor or DMA.
		14: OTHER	Count all requests driven by other processors or DMA.
		15: PREFETCH	Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the ioq_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles. 4: The mapping of interpreted bit field values to transaction types may differ across different processor model implementations of the Pentium 4 processor family. Applications that programs performance monitoring events should use the CPUID instruction to detect processor models when using this event. The logical expression that triggers this event as describe below: 5a:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)).

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5b:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5c: This event is known to ignore CPL in the current implementations of Pentium 4 and Xeon Processors Both user requests and OS requests are included in the count.
			6: An allocated entry can be a full line (64 bytes) or in individual chunks of 8 bytes.
FSB_data_ activity			This event increments once for each DRDY or DBSY event that occurs on the front side bus. The event allows selection of a specific DRDY or DBSY event.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	17H	ESCR[31:25]
	ESCR Event Mask	Bit 0: DRDY_DRV	ESCR[24:9] Count when this processor drives data onto the bus - includes writes and implicit writebacks. Asserted two processor clock cycles for partial writes and 4 processor clocks (usually in consecutive bus clocks) for full line writes.
		1: DRDY_OWN	Count when this processor reads data from the bus - includes loads and some PIC transactions. Asserted two processor clock cycles for partial reads and 4 processor clocks (usually in consecutive bus clocks) for full line reads. Count DRDY events that we drive. Count DRDY events sampled that we own.
		2: DRDY_OTHER	Count when data is on the bus but not being sampled by the processor. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
		3: DBSY_DRV	Count when this processor reserves the bus for use in the next bus cycle in order to drive data. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (in consecutive bus clocks) if we stall the bus waiting for a cache lock to complete.
		4: DBSY_OWN	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will sample. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (all one bus clock apart) if we stall the bus for some reason.

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		5:DBSY_OTHER	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will NOT sample. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		Specify edge trigger in the CCCR MSR to avoid double counting. DRDY_OWN and DRDY_OTHER are mutually exclusive; similarly for DBSY_OWN and DBSY_OTHER.
BSQ_allocation			This event counts allocations in the Bus Sequence Unit (BSQ) according to the specified mask bit encoding. The event mask bits consist of four sub-groups: ▪ Request type. ▪ Request length. ▪ Memory type. ▪ Sub-group consisting mostly of independent bits (bits 5, 6, 7, 8, 9, and 10). Specify an encoding for each sub-group.
	ESCR restrictions	MSR_BSU_ESCR0	
	Counter numbers per ESCR	ESCR0: 0, 1	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit	ESCR[24:9]
		0: REQ_TYPE0 1: REQ_TYPE1	Request type encoding (bit 0 and 1) are: 0 – Read (excludes read invalidate). 1 – Read invalidate. 2 – Write (other than writebacks). 3 – Writeback (evicted from cache). (public)
		2: REQ_LEN0 3: REQ_LEN1	Request length encoding (bit 2, 3) are: 0 – 0 chunks 1 – 1 chunks 3 – 8 chunks
		5: REQ_IO_TYPE	Request type is input or output.
		6: REQ_LOCK_ TYPE	Request type is bus lock.
		7: REQ_CACHE_ TYPE	Request type is cacheable.
		8: REQ_SPLIT_ TYPE	Request type is a bus 8-byte chunk split across 8-byte boundary.
		9: REQ_DEM_TYPE	Request type is a demand if set. Request type is HW.SW prefetch if 0.
		10: REQ_ORD_ TYPE	Request is an ordered type.

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		11: MEM_TYPE0 12: MEM_TYPE1 13: MEM_TYPE2	Memory type encodings (bit 11-13) are: 0 - UC 1 - WC 4 - WT 5 - WP 6 - WB
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specify edge trigger in CCCR to avoid double counting. 2: A writebacks to 3rd level cache from 2nd level cache counts as a separate entry, this is in additional to the entry allocated for a request to the bus. 3: A read request to WB memory type results in a request to the 64-byte sector, containing the target address, followed by a prefetch request to an adjacent sector. 4: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 0 and 1, an allocated BSQ entry includes both the demand sector and prefetched 2nd sector. 5: An allocated BSQ entry for a data chunk is any request less than 64 bytes. 6a: This event may undercount for requests of split type transactions if the data address straddled across modulo-64 byte boundary. 6b: This event may undercount for requests of read request of 16-byte operands from WC or UC address. 6c: This event may undercount WC partial requests originated from store operands that are dwords.
bsq_active_ entries			This event represents the number of BSQ entries (clipped at 15) currently active (valid) which meet the subevent mask criteria during allocation in the BSQ. Active request entries are allocated on the BSQ until de-allocated. De-allocation of an entry does not necessarily imply the request is filled. This event must be programmed in conjunction with BSQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	ESCR Event Mask		ESCR[24:9]
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the BSQ_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles.
			4: This event can be used to estimate the latency of a transaction from allocation to de-allocation in the BSQ. The latency observed by BSQ_allocation includes the latency of FSB, plus additional overhead.

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5: Additional overhead may include the time it takes to issue two requests (the sector by demand and the adjacent sector via prefetch). Since adjacent sector prefetches have lower priority that demand fetches, on a heavily used system there is a high probability that the adjacent sector prefetch will have to wait until the next bus arbitration. 6: For Pentium 4 and Xeon processors with CPUID model encoding value less than 3, this event is updated every clock. 7: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 3 or 4, this event is updated every other clock.
SSE_input_assist			This event counts the number of times an assist is requested to handle problems with input operands for SSE/SSE2/SSE3 operations; most notably denormal source operands when the DAZ bit is not set. Set bit 15 of the event mask to use this event.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	34H	ESCR[31:25]
	ESCR Event Mask	 15: ALL	ESCR[24:9] Count assists for SSE/SSE2/SSE3 μops.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: Not all requests for assists are actually taken. This event is known to overcount in that it counts requests for assists from instructions on the non-retired path that do not incur a performance penalty. An assist is actually taken only for non-bogus μops. Any appreciable counts for this event are an indication that the DAZ or FTZ bit should be set and/or the source code should be changed to eliminate the condition. 2: Two common situations for an SSE/SSE2/SSE3 operation needing an assist are: (1) when a denormal constant is used as an input and the Denormals-Are-Zero (DAZ) mode is not set, (2) when the input operand uses the underflowed result of a previous SSE/SSE2/SSE3 operation and neither the DAZ nor Flush-To-Zero (FTZ) modes are set. 3: Enabling the DAZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the first situation. Enabling the FTZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the second situation.
packed_SP_uop			This event increments for each packed single-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all µops operating on packed single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: If an instruction contains more than one packed SP µops, each packed SP µop that is specified by the event mask will be counted. 2: This metric counts instances of packed memory µops in a repeat move string.
packed_DP_uop			This event increments for each packed double-precision µop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all µops operating on packed double-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one packed DP µops, each packed DP µop that is specified by the event mask will be counted.
scalar_SP_uop			This event increments for each scalar single-precision µop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all µops operating on scalar single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one scalar SP µops, each scalar SP µop that is specified by the event mask will be counted.
scalar_DP_uop			This event increments for each scalar double-precision µop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0EH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all µops operating on scalar double-precision operands.
	CCCR Select	01H	CCCR[15:13]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		If an instruction contains more than one scalar DP μops, each scalar DP μop that is specified by the event mask is counted.
64bit_MMX_uop			This event increments for each MMX instruction, which operate on 64-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 64- bit SIMD integer operands in memory or MMX registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 64-bit MMX μops, each 64-bit MMX μop that is specified by the event mask will be counted.
128bit_MMX_uop			This event increments for each integer SIMD SSE2 instruction, which operate on 128-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	1AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 128-bit SIMD integer operands in memory or XMM registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 128-bit MMX μops, each 128-bit MMX μop that is specified by the event mask will be counted.
x87_FP_uop			This event increments for each x87 floating-point μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all x87 FP μops.
	CCCR Select	01H	CCCR[15:13]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		1: If an instruction contains more than one x87 FP μops, each x87 FP μop that is specified by the event mask will be counted. 2: This event does not count x87 FP μop for load, store, move between registers.
TC_misc			This event counts miscellaneous events detected by the TC. The counter will count twice for each occurrence.
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	06H	ESCR[31:25]
	CCCR Select	01H	CCCR[15:13]
	ESCR Event Mask	 Bit 4: FLUSH	ESCR[24:9] Number of flushes
global_power _events			This event accumulates the time during which a processor is not stopped.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	013H	ESCR[31:25]
	ESCR Event Mask	Bit 0: Running	ESCR[24:9] The processor is active (includes the handling of HLT STPCLK and throttling.
	CCCR Select	06H	CCCR[15:13]
tc_ms_xfer			This event counts the number of times that uop delivery changed from TC to MS ROM.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: CISC	ESCR[24:9] A TC to MS transfer occurred.
	CCCR Select	0H	CCCR[15:13]
uop_queue_ writes			This event counts the number of valid uops written to the uop queue. Specify one or more mask bits to select the source type of writes.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FROM_TC_ BUILD 1: FROM_TC_ DELIVER 2: FROM_ROM	ESCR[24:9] The uops being written are from TC build mode. The uops being written are from TC deliver mode. The uops being written are from microcode ROM.
	CCCR Select	0H	CCCR[15:13]
retired_mispred _branch_type			This event counts retiring mispredicted branches by type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL	ESCR[24:9] Conditional jumps. Indirect call branches.
		3: RETURN 4: INDIRECT	Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		This event may overcount conditional branches if: ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
retired_branch _type			This event counts retiring branches by type. Specify one or more mask bits to qualify the branch by its type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	04H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL 3: RETURN 4: INDIRECT	ESCR[24:9] Conditional jumps. Direct or indirect calls. Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		This event may overcount conditional branches if : ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
resource_stall			This event monitors the occurrence or latency of stalls in the Allocator.
	ESCR restrictions	MSR_ALF_ESCR0 MSR_ALF_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[30:25]
	Event Masks	Bit 5: SBFULL	ESCR[24:9] A Stall due to lack of store buffers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
WC_Buffer			This event counts Write Combining Buffer operations that are selected by the event mask.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[30:25]
	Event Masks	Bit 0: WCB_EVICTS	ESCR[24:9] WC Buffer evictions of all causes.
		1: WCB_FULL_EVICT	WC Buffer eviction: no WC buffer is available.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		This event is useful for detecting the subset of 64K aliasing cases that are more costly (i.e. 64K aliasing cases involving stores) as long as there are no significant contributions due to write combining buffer full or hit-modified conditions.
b2b_cycles			This event can be configured to count the number back-to-back bus cycles using sub-event mask bits 1 through 6.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	016H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]

Table 19-29. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
bnr			This event can be configured to count bus not ready conditions using sub-event mask bits 0 through 2.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	08H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
snoop			This event can be configured to count snoop hit modified bus traffic using sub-event mask bits 2, 6 and 7.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
Response			This event can be configured to count different types of responses using sub-event mask bits 1,2, 8, and 9.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	04H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.

Table 19-30. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
front_end_event			This event counts the retirement of tagged μops, which are specified through the front-end tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	08H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	Selected ESCRs and/or MSR_TC_ PRECISE_EVENT	See list of metrics supported by Front_end tagging in Table A-3
execution_event			This event counts the retirement of tagged μops, which are specified through the execution tagging mechanism. The event mask allows from one to four types of μops to be specified as either bogus or non-bogus μops to be tagged.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS0 1: NBOGUS1 2: NBOGUS2 3: NBOGUS3 4: BOGUS0 5: BOGUS1 6: BOGUS2 7: BOGUS3	ESCR[24:9] The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Each of the 4 slots to specify the bogus/non-bogus μops must be coordinated with the 4 TagValue bits in the ESCR (for example, NBOGUS0 must accompany a '1' in the lowest bit of the TagValue field in ESCR, NBOGUS1 must accompany a '1' in the next but lowest bit of the TagValue field).
	Can Support PEBS	Yes	

Table 19-30. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Require Additional MSRs for tagging	An ESCR for an upstream event	See list of metrics supported by execution tagging in Table A-4.
replay_event			This event counts the retirement of tagged μops, which are specified through the replay tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Supports counting tagged μops with additional MSRs.
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	IA32_PEBS_ ENABLE MSR_PEBS_ MATRIX_VERT Selected ESCR	See list of metrics supported by replay tagging in Table A-5.
instr_retired			This event counts instructions that are retired during a clock cycle. Mask bits specify bogus or non-bogus (and whether they are tagged using the front-end tagging mechanism).
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUSNTAG 1: NBOGUSTAG 2: BOGUSNTAG 3: BOGUSTAG	ESCR[24:9] Non-bogus instructions that are not tagged. Non-bogus instructions that are tagged. Bogus instructions that are not tagged. Bogus instructions that are tagged.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		1: The event count may vary depending on the microarchitectural states of the processor when the event detection is enabled. 2: The event may count more than once for some instructions with complex uop flows and were interrupted before retirement.

Table 19-30. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Can Support PEBS	No	
uops_retired			This event counts µops that are retired during a clock cycle. Mask bits specify bogus or non-bogus.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked µops are not bogus. The marked µops are bogus.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		P6: EMON_UOPS_RETIRED
	Can Support PEBS	No	
uop_type			This event is used in conjunction with the front-end at-retirement mechanism to tag load and store µops.
	ESCR restrictions	MSR_RAT_ESCR0 MSR_RAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 1: TAGLOADS 2: TAGSTORES	ESCR[24:9] The µop is a load operation. The µop is a store operation.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Setting the TAGLOADS and TAGSTORES mask bits does not cause a counter to increment. They are only used to tag uops.
	Can Support PEBS	No	
branch_retired			This event counts the retirement of a branch. Specify one or more mask bits to select any combination of taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 18-67 for the addresses of the ESCR MSRs
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 18-67.
	ESCR Event Select	06H	ESCR[31:25]

Table 19-30. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch not-taken predicted Branch not-taken mispredicted Branch taken predicted Branch taken mispredicted
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
mispred_branch_retired			This event represents the retirement of mispredicted branch instructions.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS	ESCR[24:9] The retired instruction is not bogus.
	CCCR Select	04H	CCCR[15:13]
	Can Support PEBS	No	
x87_assist			This event counts the retirement of x87 instructions that required special handling. Specifies one or more event mask bits to select the type of assistance.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FPSU 1: FPSO 2: POAO 3: POAU 4: PREA	ESCR[24:9] Handle FP stack underflow. Handle FP stack overflow. Handle x87 output overflow. Handle x87 output underflow. Handle x87 input assist.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-30. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
machine_clear			This event increments according to the mask bit specified while the entire pipeline of the machine is cleared. Specify one of the mask bit to select the cause.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 0: CLEAR 2: MOCLEAR 6: SMCLEAR	ESCR[24:9] Counts for a portion of the many cycles while the machine is cleared for any cause. Use Edge triggering for this bit only to get a count of occurrence versus a duration. Increments each time the machine is cleared due to memory ordering issues. Increments each time the machine is cleared due to self-modifying code issues.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-31. Intel NetBurst® Microarchitecture Model-Specific Performance Monitoring Events (For Model Encoding 3, 4 or 6)

Event Name	Event Parameters	Parameter Value	Description
instr_completed			This event counts instructions that have completed and retired during a clock cycle. Mask bits specify whether the instruction is bogus or non-bogus and whether they are:
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	07H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] Non-bogus instructions Bogus instructions
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		This metric differs from instr_retired, since it counts instructions completed, rather than the number of times that instructions started.
	Can Support PEBS	No	

Table 19-32. List of Metrics Available for Front_end Tagging (For Front_end Event Only)

Front-end metric[1]	MSR_TC_PRECISE_EVENT MSR Bit field	Additional MSR	Event mask value for Front_end_event
memory_loads	None	Set TAGLOADS bit in ESCR corresponding to event Uop_Type.	NBOGUS
memory_stores	None	Set TAGSTORES bit in the ESCR corresponding to event Uop_Type.	NBOGUS

NOTES:

1. There may be some undercounting of front end events when there is an overflow or underflow of the floating point stack.

Table 19-33. List of Metrics Available for Execution Tagging (For Execution Event Only)

Execution metric	Upstream ESCR	TagValue in Upstream ESCR	Event mask value for execution_event
packed_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_SP_uop.	1	NBOGUS0
packed_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_DP_uop.	1	NBOGUS0
scalar_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_SP_uop.	1	NBOGUS0
scalar_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_DP_uop.	1	NBOGUS0
128_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 128_bit_MMX_uop.	1	NBOGUS0
64_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 64_bit_MMX_uop.	1	NBOGUS0
X87_FP_retired	Set ALL bit in event mask, TagUop bit in ESCR of x87_FP_uop.	1	NBOGUS0
X87_SIMD_memory_moves_retired	Set ALLP0, ALLP2 bits in event mask, TagUop bit in ESCR of X87_SIMD_ moves_uop.	1	NBOGUS0

Table 19-34. List of Metrics Available for Replay Tagging (For Replay Event Only)

Replay metric[1]	IA32_PEBS_ENABLE Field to Set	MSR_PEBS_MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
1stL_cache_load_miss_retired	Bit 0, Bit 24, Bit 25	Bit 0	None	NBOGUS
2ndL_cache_load_miss_retired[2]	Bit 1, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_load_miss_retired	Bit 2, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_store_miss_retired	Bit 2, Bit 24, Bit 25	Bit 1	None	NBOGUS
DTLB_all_miss_retired	Bit 2, Bit 24, Bit 25	Bit 0, Bit 1	None	NBOGUS
Tagged_mispred_branch	Bit 15, Bit 16, Bit 24, Bit 25	Bit 4	None	NBOGUS

Table 19-34. List of Metrics Available for Replay Tagging (For Replay Event Only) (Contd.)

Replay metric[1]	IA32_PEBS_ ENABLE Field to Set	MSR_PEBS_ MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
MOB_load _replay_retired[3]	Bit 9, Bit 24, Bit 25	Bit 0	Select MOB_load_replay event and set PARTIAL_DATA and UNALGN_ADDR bit.	NBOGUS
split_load_retired	Bit 10, Bit 24, Bit 25	Bit 0	Select load_port_replay event with the MSR_SAAT_ESCR1 MSR and set the SPLIT_LD mask bit.	NBOGUS
split_store_retired	Bit 10, Bit 24, Bit 25	Bit 1	Select store_port_replay event with the MSR_SAAT_ESCR0 MSR and set the SPLIT_ST mask bit.	NBOGUS

NOTES:

1. Certain kinds of μops cannot be tagged. These include I/O operations, UC and locked accesses, returns, and far transfers.
2. 2nd-level misses retired does not count all 2nd-level misses. It only includes those references that are found to be misses by the fast detection logic and not those that are later found to be misses.
3. While there are several causes for a MOB replay, the event counted with this event mask setting is the case where the data from a load that would otherwise be forwarded is not an aligned subset of the data from a preceding store.

Table 19-35. Event Mask Qualification for Logical Processors

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	BPU_fetch_request	Bit 0: TCMISS	TS
Non-Retirement	BSQ_allocation	Bit 0: REQ_TYPE0 1: REQ_TYPE1 2: REQ_LEN0 3: REQ_LEN1 5: REQ_IO_TYPE 6: REQ_LOCK_TYPE 7: REQ_CACHE_TYPE 8: REQ_SPLIT_TYPE 9: REQ_DEM_TYPE 10: REQ_ORD_TYPE 11: MEM_TYPE0 12: MEM_TYPE1 13: MEM_TYPE2	 TS TS TS TS TS TS TS TS TS TS TS TS TS
Non-Retirement	BSQ_cache_reference	Bit 0: RD_2ndL_HITS 1: RD_2ndL_HITE 2: RD_2ndL_HITM 3: RD_3rdL_HITS 4: RD_3rdL_HITE 5: RD_3rdL_HITM 6: WR_2ndL_HIT 7: WR_3rdL_HIT 8: RD_2ndL_MISS 9: RD_3rdL_MISS 10: WR_2ndL_MISS 11: WR_3rdL_MISS	 TS TS TS TS TS TS TS TS TS TS TS TS
Non-Retirement	memory_cancel	Bit 2: ST_RB_FULL 3: 64K_CONF	 TS TS
Non-Retirement	SSE_input_assist	Bit 15: ALL	TI
Non-Retirement	64bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	packed_DP_uop	Bit 15: ALL	TI
Non-Retirement	packed_SP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_DP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_SP_uop	Bit 15: ALL	TI
Non-Retirement	128bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	x87_FP_uop	Bit 15: ALL	TI

Table 19-35. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	x87_SIMD_moves_uop	Bit	
		3: ALLP0	TI
		4: ALLP2	TI
Non-Retirement	FSB_data_activity	Bit	
		0: DRDY_DRV	TI
		1: DRDY_OWN	TI
		2: DRDY_OTHER	TI
		3: DBSY_DRV	TI
		4: DBSY_OWN	TI
		5: DBSY_OTHER	TI
Non-Retirement	IOQ_allocation	Bit	
		0: ReqA0	TS
		1: ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS
		13: OWN	TS
		14: OTHER	TS
		15: PREFETCH	TS
Non-Retirement	IOQ_active_entries	Bit	TS
		0: ReqA0	
		1:ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS

Table 19-35. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		13: OWN 14: OTHER 15: PREFETCH	TS TS TS
Non-Retirement	global_power_events	Bit 0: RUNNING	TS
Non-Retirement	ITLB_reference	Bit 0: HIT 1: MISS 2: HIT_UC	TS TS TS
Non-Retirement	MOB_load_replay	Bit 1: NO_STA 3: NO_STD 4: PARTIAL_DATA 5: UNALGN_ADDR	TS TS TS TS
Non-Retirement	page_walk_type	Bit 0: DTMISS 1: ITMISS	TI TI
Non-Retirement	uop_type	Bit 1: TAGLOADS	TS
		2: TAGSTORES	TS
Non-Retirement	load_port_replay	Bit 1: SPLIT_LD	TS
Non-Retirement	store_port_replay	Bit 1: SPLIT_ST	TS
Non-Retirement	memory_complete	Bit 0: LSC 1: SSC 2: USC 3: ULC	TS TS TS TS
Non-Retirement	retired_mispred_branch_type	Bit 0: UNCONDITIONAL 1: CONDITIONAL 2: CALL 3: RETURN 4: INDIRECT	TS TS TS TS TS
Non-Retirement	retired_branch_type	Bit 0: UNCONDITIONAL 1: CONDITIONAL 2: CALL 3: RETURN 4: INDIRECT	TS TS TS TS TS

Table 19-35. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	tc_ms_xfer	Bit	
		0: CISC	TS
Non-Retirement	tc_misc	Bit	
		4: FLUSH	TS
Non-Retirement	TC_deliver_mode	Bit	
		0: DD	TI
		1: DB	TI
		2: DI	TI
		3: BD	TI
		4: BB	TI
		5: BI	TI
		6: ID	TI
		7: IB	TI
Non-Retirement	uop_queue_writes	Bit	
		0: FROM_TC_BUILD	TS
		1: FROM_TC_DELIVER	TS
		2: FROM_ROM	TS
Non-Retirement	resource_stall	Bit 5: SBFULL	TS
Non-Retirement	WC_Buffer	Bit	TI
		0: WCB_EVICTS	TI
		1: WCB_FULL_EVICT	TI
		2: WCB_HITM_EVICT	TI
At Retirement	instr_retired	Bit	
		0: NBOGUSNTAG	TS
		1: NBOGUSTAG	TS
		2: BOGUSNTAG	TS
		3: BOGUSTAG	TS
At Retirement	machine_clear	Bit	
		0: CLEAR	TS
		2: MOCLEAR	TS
		6: SMCCLEAR	TS
At Retirement	front_end_event	Bit	
		0: NBOGUS	TS
		1: BOGUS	TS
At Retirement	replay_event	Bit	
		0: NBOGUS	TS
		1: BOGUS	TS
At Retirement	execution_event	Bit	
		0: NONBOGUS0	TS
		1: NONBOGUS1	TS

Table 19-35. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		2: NONBOGUS2 3: NONBOGUS3 4: BOGUS0 5: BOGUS1 6: BOGUS2 7: BOGUS3	TS TS TS TS TS TS
At Retirement	x87_assist	Bit 0: FPSU 1: FPSO 2: POAO 3: POAU 4: PREA	 TS TS TS TS TS
At Retirement	branch_retired	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	 TS TS TS TS
At Retirement	mispred_branch_retired	Bit 0: NBOGUS	TS
At Retirement	uops_retired	Bit 0: NBOGUS 1: BOGUS	 TS TS
At Retirement	instr_completed	Bit 0: NBOGUS 1: BOGUS	 TS TS

19.17 PERFORMANCE MONITORING EVENTS FOR INTEL® PENTIUM® M PROCESSORS

The Pentium M processor's performance-monitoring events are based on monitoring events for the P6 family of processors. All of these performance events are model specific for the Pentium M processor and are not available in this form in other processors. Table 19-36 lists the Performance-Monitoring events that were added in the Pentium M processor.

Table 19-36. Performance Monitoring Events on Intel® Pentium® M Processors

Name	Hex Values	Descriptions
Power Management		
EMON_EST_TRANS	58H	Number of Enhanced Intel SpeedStep technology transitions: Mask = 00H - All transitions Mask = 02H - Only Frequency transitions
EMON_THERMAL_TRIP	59H	Duration/Occurrences in thermal trip; to count number of thermal trips: bit 22 in PerfEvtSel0/1 needs to be set to enable edge detect.
BPU		
BR_INST_EXEC	88H	Branch instructions that were executed (not necessarily retired).
BR_MISSP_EXEC	89H	Branch instructions executed that were mispredicted at execution.
BR_BAC_MISSP_EXEC	8AH	Branch instructions executed that were mispredicted at front end (BAC).
BR_CND_EXEC	8BH	Conditional branch instructions that were executed.
BR_CND_MISSP_EXEC	8CH	Conditional branch instructions executed that were mispredicted.
BR_IND_EXEC	8DH	Indirect branch instructions executed.
BR_IND_MISSP_EXEC	8EH	Indirect branch instructions executed that were mispredicted.
BR_RET_EXEC	8FH	Return branch instructions executed.
BR_RET_MISSP_EXEC	90H	Return branch instructions executed that were mispredicted at execution.
BR_RET_BAC_MISSP_EXEC	91H	Return branch instructions executed that were mispredicted at front end (BAC).
BR_CALL_EXEC	92H	CALL instruction executed.
BR_CALL_MISSP_EXEC	93H	CALL instruction executed and miss predicted.
BR_IND_CALL_EXEC	94H	Indirect CALL instructions executed.
Decoder		
EMON_SIMD_INSTR_RETIRED	CEH	Number of retired MMX instructions.
EMON_SYNCH_UOPS	D3H	Sync micro-ops
EMON_ESP_UOPS	D7H	Total number of micro-ops
EMON_FUSED_UOPS_RET	DAH	Number of retired fused micro-ops: Mask = 0 - Fused micro-ops Mask = 1 - Only load+Op micro-ops Mask = 2 - Only std+sta micro-ops
EMON_UNFUSION	DBH	Number of unfusion events in the ROB, happened on a FP exception to a fused μop.
Prefetcher		
EMON_PREF_RQSTS_UP	F0H	Number of upward prefetches issued.
EMON_PREF_RQSTS_DN	F8H	Number of downward prefetches issued.

A number of P6 family processor performance monitoring events are modified for the Pentium M processor. Table 19-37 lists the performance monitoring events that were changed in the Pentium M processor, and differ from performance monitoring events for the P6 family of processors.

Table 19-37. Performance Monitoring Events Modified on Intel® Pentium® M Processors

<table>
<tr><th>Name</th><th>Hex Values</th><th colspan="2">Descriptions</th></tr>
<tr><td>CPU_CLK_UNHALTED</td><td>79H</td><td colspan="2">Number of cycles during which the processor is not halted, and not in a thermal trip.</td></tr>
<tr><td>EMON_SSE_SSE2_INST_RETIRED</td><td>D8H</td><td colspan="2">Streaming SIMD Extensions Instructions Retired:
Mask = 0 – SSE packed single and scalar single
Mask = 1 – SSE scalar-single
Mask = 2 – SSE2 packed-double
Mask = 3 – SSE2 scalar-double</td></tr>
<tr><td>EMON_SSE_SSE2_COMP_INST_RETIRED</td><td>D9H</td><td colspan="2">Computational SSE Instructions Retired:
Mask = 0 – SSE packed single
Mask = 1 – SSE Scalar-single
Mask = 2 – SSE2 packed-double
Mask = 3 – SSE2 scalar-double</td></tr>
<tr><td>L2_LD</td><td>29H</td><td>L2 data loads</td><td rowspan="4">Mask[0] = 1 – count I state lines
Mask[1] = 1 – count S state lines
Mask[2] = 1 – count E state lines
Mask[3] = 1 – count M state lines
Mask[5:4]:
00H – Excluding hardware-prefetched lines
01H - Hardware-prefetched lines only
02H/03H – All (HW-prefetched lines and non HW -- Prefetched lines)</td></tr>
<tr><td>L2_LINES_IN</td><td>24H</td><td>L2 lines allocated</td></tr>
<tr><td>L2_LINES_OUT</td><td>26H</td><td>L2 lines evicted</td></tr>
<tr><td>L2_M_LINES_OUT</td><td>27H</td><td>Lw M-state lines evicted</td></tr>
</table>

19.18 P6 FAMILY PROCESSOR PERFORMANCE-MONITORING EVENTS

Table 19-38 lists the events that can be counted with the performance-monitoring counters and read with the RDPMC instruction for the P6 family processors. The unit column gives the microarchitecture or bus unit that produces the event; the event number column gives the hexadecimal number identifying the event; the mnemonic event name column gives the name of the event; the unit mask column gives the unit mask required (if any); the description column describes the event; and the comments column gives additional information about the event.

All of these performance events are model specific for the P6 family processors and are not available in this form in the Pentium 4 processors or the Pentium processors. Some events (such as those added in later generations of the P6 family processors) are only available in specific processors in the P6 family. All performance event encodings not listed in Table 19-38 are reserved and their use will result in undefined counter results.

See the end of the table for notes related to certain entries in the table.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	43H	DATA_MEM_REFS	00H	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. The internal logic counts not only memory loads and stores, but also internal retries. 80-bit floating-point accesses are double counted, since they are decomposed into a 16-bit exponent load and a 64-bit mantissa load. Memory accesses are only counted when they are actually performed (such as a load that gets squashed because a previous cache miss is outstanding to the same address, and which finally gets performed, is only counted once). Does not include I/O accesses, or other nonmemory accesses.	
	45H	DCU_LINES_IN	00H	Total lines allocated in DCU.	
	46H	DCU_M_LINES_IN	00H	Number of M state lines allocated in DCU.	
	47H	DCU_M_LINES_OUT	00H	Number of M state lines evicted from DCU. This includes evictions via snoop HITM, intervention or replacement.	
	48H	DCU_MISS_OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding, incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. Uncacheable requests are excluded. Read-for-ownerships are counted, as well as line fills, invalidates, and stores.	An access that also misses the L2 is short-changed by 2 cycles (i.e., if counts N cycles, should be N+2 cycles). Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and noncacheable, including UC fetches.	
	81H	IFU_IFETCH_MISS	00H	Number of instruction fetch misses All instruction fetches that do not hit the IFU (i.e., that produce memory requests). This includes UC accesses.	
	85H	ITLB_MISS	00H	Number of ITLB misses.	
	86H	IFU_MEM_STALL	00H	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults, and other minor stalls.	
	87H	ILD_STALL	00H	Number of cycles that the instruction length decoder is stalled.	
L2 Cache[1]	28H	L2_IFETCH	MESI 0FH	Number of L2 instruction fetches. This event indicates that a normal instruction fetch was received by the L2.	

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				The count includes only L2 cacheable instruction fetches; it does not include UC instruction fetches. It does not include ITLB miss accesses.	
	29H	L2_LD	MESI 0FH	Number of L2 data loads. This event indicates that a normal, unlocked, load memory access was received by the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It does include L2 cacheable TLB miss memory accesses.	
	2AH	L2_ST	MESI 0FH	Number of L2 data stores. This event indicates that a normal, unlocked, store memory access was received by the L2. it indicates that the DCU sent a read-for-ownership request to the L2. It also includes Invalid to Modified requests sent by the DCU to the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It includes TLB miss memory accesses.	
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_INM	00H	Number of modified lines allocated in the L2.	
	27H	L2_M_LINES_ OUTM	00H	Number of modified lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Total number of L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the L2 cache data bus was busy.	
	23H	L2_DBUS_BUSY_ RD	00H	Number of cycles during which the data bus was busy transferring read data from L2 to the processor.	
External Bus Logic (EBL)[2]	62H	BUS_DRDY_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY# is asserted. Utilization of the external system data bus during data transfers.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY#. Unit Mask = 20H counts in processor clocks when any agent is driving DRDY#.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	63H	BUS_LOCK_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK# is asserted on the external system bus.[3]	Always counts in processor clocks.
	60H	BUS_REQ_ OUTSTANDING	00H (Self)	Number of bus requests outstanding. This counter is incremented by the number of cacheable read bus requests outstanding in any given cycle.	Counts only DCU full-line cacheable reads, not RFOs, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of burst read transactions.	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of completed read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of completed write back transactions.	
	68H	BUS_TRAN_ IFETCH	00H (Self) 20H (Any)	Number of completed instruction fetch transactions.	
	69H	BUS_TRAN_INVA L	00H (Self) 20H (Any)	Number of completed invalidate transactions.	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	Number of completed partial write transactions.	
	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of completed partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of completed I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of completed deferred transactions.	

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of completed burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all completed bus transactions. Address bus utilization can be calculated knowing the minimum address bus occupancy. Includes special cycles, etc.	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	Number of completed memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR# pin.	
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow. ▪ If the PC bit is clear, the processor toggles the BPM*i* pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
					▪ If the PC bit is clear, the processor toggles the BPM*i*pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired. Excludes floating-point computational operations that cause traps or assists. Includes floating-point computational operations executed by the assist handler. Includes internal sub-operations for complex floating-point instructions like transcendentals. Excludes floating-point loads and stores.	Counter 0 only.
	10H	FP_COMP_OPS_ EXE	00H	Number of computational floating-point operations executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This number does not include the number of cycles, but the number of operations. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only. This event includes counts due to speculative execution.
	12H	MUL	00H	Number of multiplies. This count includes integer as well as FP multiplies and is speculative.	Counter 1 only.
	13H	DIV	00H	Number of divides. This count includes integer as well as FP divides and is speculative.	Counter 1 only.
	14H	CYCLES_DIV_ BUSY	00H	Number of cycles during which the divider is busy, and cannot accept new divides. This includes integer and FP divides, FPREM, FPSQRT, etc. and is speculative.	Counter 0 only.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Memory Ordering	03H	LD_BLOCKS	00H	Number of load operations delayed due to store buffer blocks. Includes counts caused by preceding stores whose addresses are unknown, preceding stores whose addresses are known but whose data is unknown, and preceding stores that conflicts with the load but which incompletely overlap the load.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles. Incremented every cycle the store buffer is draining. Draining is caused by serializing operations like CPUID, synchronizing operations like XCHG, interrupt acknowledgment, as well as other conditions (such as cache flushing).	
	05H	MISALIGN_ MEM_REF	00H	Number of misaligned data memory references. Incremented by 1 every cycle, during which either the processor's load or store pipeline dispatches a misaligned µop. Counting is performed if it is the first or second half, or if it is blocked, squashed, or missed. In this context, misaligned means crossing a 64-bit boundary.	MISALIGN_MEM_ REF is only an approximation to the true number of misaligned memory references. The value returned is roughly proportional to the number of misaligned memory accesses (the size of the problem).
	07H	EMON_KNI_PREF _DISPATCHED	 00H 01H 02H 03H	Number of Streaming SIMD extensions prefetch/weakly-ordered instructions dispatched (speculative prefetches are included in counting): 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
	4BH	EMON_KNI_PREF _MISS	 00H 01H 02H 03H	Number of prefetch/weakly-ordered instructions that miss all caches: 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
Instruction Decoding and Retirement	C0H	INST_RETIRED	00H	Number of instructions retired.	A hardware interrupt received during/after the last iteration of the REP STOS flow causes the counter to undercount by 1 instruction.
					An SMI received while executing a HLT instruction will cause the performance counter to not count the RSM instruction and undercount by 1.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	C2H	UOPS_RETIRED	00H	Number of µops retired.	
	D0H	INST_DECODED	00H	Number of instructions decoded.	
	D8H	EMON_KNI_INST_ RETIRED	 00H 01H	Number of Streaming SIMD extensions retired: 0: packed & scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
	D9H	EMON_KNI_ COMP_ INST_RET	 00H 01H	Number of Streaming SIMD extensions computation instructions retired: 0: packed and scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
Interrupts	C8H	HW_INT_RX	00H	Number of hardware interrupts received.	
	C6H	CYCLES_INT_ MASKED	00H	Number of processor cycles for which interrupts are disabled.	
	C7H	CYCLES_INT_ PENDING_ AND_MASKED	00H	Number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_ RETIRED	00H	Number of branch instructions retired.	
	C5H	BR_MISS_PRED_ RETIRED	00H	Number of mispredicted branches retired.	
	C9H	BR_TAKEN_ RETIRED	00H	Number of taken branches retired.	
	CAH	BR_MISS_PRED_ TAKEN_RET	00H	Number of taken mispredictions branches retired.	
	E0H	BR_INST_ DECODED	00H	Number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Number of branches for which the BTB did not produce a prediction.	
	E4H	BR_BOGUS	00H	Number of bogus branches.	
	E6H	BACLEARS	00H	Number of times BACLEAR is asserted. This is the number of times that a static branch prediction was made, in which the branch decoder decided to make a branch prediction because the BTB did not.	
Stalls	A2H	RESOURCE_ STALLS	00H	Incremented by 1 during every cycle for which there is a resource related stall. Includes register renaming buffer entries, memory buffer entries.	

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				Does not include stalls due to bus queue full, too many cache misses, etc. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls. This includes flag partial stalls.	
Segment Register Loads	06H	SEGMENT_REG_ LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX Unit	B0H	MMX_INSTR_ EXEC	00H	Number of MMX Instructions Executed.	Available in Intel Celeron, Pentium II and Pentium II Xeon processors only. Does not account for MOVQ and MOVD stores from register to memory.
	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX Saturating Instructions Executed.	Available in Pentium II and Pentium III processors only.
	B2H	MMX_UOPS_ EXEC	0FH	Number of MMX µops Executed.	Available in Pentium II and Pentium III processors only.
	B3H	MMX_INSTR_ TYPE_EXEC	01H 02H 04H 08H 10H 20H	MMX packed multiply instructions executed. MMX packed shift instructions executed. MMX pack operation instructions executed. MMX unpack operation instructions executed. MMX packed logical instructions executed. MMX packed arithmetic instructions executed.	Available in Pentium II and Pentium III processors only.
	CCH	FP_MMX_TRANS	00H 01H	Transitions from MMX instruction to floating-point instructions. Transitions from floating-point instructions to MMX instructions.	Available in Pentium II and Pentium III processors only.
	CDH	MMX_ASSIST	00H	Number of MMX Assists (that is, the number of EMMS instructions executed).	Available in Pentium II and Pentium III processors only.
	CEH	MMX_INSTR_RET	00H	Number of MMX Instructions Retired.	Available in Pentium II processors only.
Segment Register Renaming	D4H	SEG_RENAME_ STALLS		Number of Segment Register Renaming Stalls:	Available in Pentium II and Pentium III processors only.

Table 19-38. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
			02H	Segment register ES	
			04H	Segment register DS	
			08H	Segment register FS	
			0FH	Segment register FS	
				Segment registers ES + DS + FS + GS	
	D5H	SEG_REG_ RENAMES		Number of Segment Register Renames:	Available in Pentium II and Pentium III processors only.
			01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register FS	
			0FH	Segment registers ES + DS + FS + GS	
	D6H	RET_SEG_ RENAMES	00H	Number of segment register rename events retired.	Available in Pentium II and Pentium III processors only.

NOTES:

1. Several L2 cache events, where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers. The lower 4 bits of the Unit Mask field are used in conjunction with L2 events to indicate the cache state or cache states involved.

   The P6 family processors identify cache states using the "MESI" protocol and consequently each bit in the Unit Mask field represents one of the four states: UMSK[3] = M (8H) state, UMSK[2] = E (4H) state, UMSK[1] = S (2H) state, and UMSK[0] = I (1H) state. UMSK[3:0] = MESI" (FH) should be used to collect data for all states; UMSK = 0H, for the applicable events, will result in nothing being counted.

2. All of the external bus logic (EBL) events, except where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers.

   Bit 5 of the UMSK field is used in conjunction with the EBL events to indicate whether the processor should count transactions that are self- generated (UMSK[5] = 0) or transactions that result from any processor on the bus (UMSK[5] = 1).

3. L2 cache locks, so it is possible to have a zero count.

19.19 PENTIUM PROCESSOR PERFORMANCE-MONITORING EVENTS

Table 19-39 lists the events that can be counted with the performance-monitoring counters for the Pentium processor. The Event Number column gives the hexadecimal code that identifies the event and that is entered in the ES0 or ES1 (event select) fields of the CESR MSR. The Mnemonic Event Name column gives the name of the event, and the Description and Comments columns give detailed descriptions of the events. Most events can be counted with either counter 0 or counter 1; however, some events can only be counted with only counter 0 or only counter 1 (as noted).

NOTE

The events in the table that are shaded are implemented only in the Pentium processor with MMX technology.

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters

Event Num.	Mnemonic Event Name	Description	Comments
00H	DATA_READ	Number of memory data reads (internal data cache hit and miss combined).	Split cycle reads are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
01H	DATA_WRITE	Number of memory data writes (internal data cache hit and miss combined); I/O not included.	Split cycle writes are counted individually. These events may occur at a maximum of two per clock. I/O is not included.
0H2	DATA_TLB_MISS	Number of misses to the data cache translation look-aside buffer.	
03H	DATA_READ_MISS	Number of memory read accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
04H	DATA WRITE MISS	Number of memory write accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
05H	WRITE_HIT_TO_ M-_OR_E-STATE_LINES	Number of write hits to exclusive or modified lines in the data cache.	These are the writes that may be held up if EWBE# is inactive. These events may occur a maximum of two per clock.
06H	DATA_CACHE_ LINES_ WRITTEN_BACK	Number of dirty lines (all) that are written back, regardless of the cause.	Replacements and internal and external snoops can all cause writeback and are counted.
07H	EXTERNAL_ SNOOPS	Number of accepted external snoops whether they hit in the code cache or data cache or neither.	Assertions of EADS# outside of the sampling interval are not counted, and no internal snoops are counted.
08H	EXTERNAL_DATA_ CACHE_SNOOP_ HITS	Number of external snoops to the data cache.	Snoop hits to a valid line in either the data cache, the data line fill buffer, or one of the write back buffers are all counted as hits.
09H	MEMORY ACCESSES IN BOTH PIPES	Number of data memory reads or writes that are paired in both pipes of the pipeline.	These accesses are not necessarily run in parallel due to cache misses, bank conflicts, etc.
0AH	BANK CONFLICTS	Number of actual bank conflicts.	
0BH	MISALIGNED DATA MEMORY OR I/O REFERENCES	Number of memory or I/O reads or writes that are misaligned.	A 2- or 4-byte access is misaligned when it crosses a 4-byte boundary; an 8-byte access is misaligned when it crosses an 8-byte boundary. Ten byte accesses are treated as two separate accesses of 8 and 2 bytes each.
0CH	CODE READ	Number of instruction reads; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0DH	CODE TLB MISS	Number of instruction reads that miss the code TLB whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0EH	CODE CACHE MISS	Number of instruction reads that miss the internal code cache; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
0FH	ANY SEGMENT REGISTER LOADED	Number of writes into any segment register in real or protected mode including the LDTR, GDTR, IDTR, and TR.	Segment loads are caused by explicit segment register load instructions, far control transfers, and task switches. Far control transfers and task switches causing a privilege level change will signal this event twice. Interrupts and exceptions may initiate a far control transfer.
10H	Reserved		
11H	Reserved		
12H	Branches	Number of taken and not taken branches, including: conditional branches, jumps, calls, returns, software interrupts, and interrupt returns.	Also counted as taken branches are serializing instructions, VERR and VERW instructions, some segment descriptor loads, hardware interrupts (including FLUSH#), and programmatic exceptions that invoke a trap or fault handler. The pipe is not necessarily flushed. The number of branches actually executed is measured, not the number of predicted branches.
13H	BTB_HITS	Number of BTB hits that occur.	Hits are counted only for those instructions that are actually executed.
14H	TAKEN_BRANCH_ OR_BTB_HIT	Number of taken branches or BTB hits that occur.	This event type is a logical OR of taken branches and BTB hits. It represents an event that may cause a hit in the BTB. Specifically, it is either a candidate for a space in the BTB or it is already in the BTB.
15H	PIPELINE FLUSHES	Number of pipeline flushes that occur Pipeline flushes are caused by BTB misses on taken branches, mispredictions, exceptions, interrupts, and some segment descriptor loads.	The counter will not be incremented for serializing instructions (serializing instructions cause the prefetch queue to be flushed but will not trigger the Pipeline Flushed event counter) and software interrupts (software interrupts do not flush the pipeline).
16H	INSTRUCTIONS_ EXECUTED	Number of instructions executed (up to two per clock).	Invocations of a fault handler are considered instructions. All hardware and software interrupts and exceptions will also cause the count to be incremented. Repeat prefixed string instructions will only increment this counter once despite the fact that the repeat loop executes the same instruction multiple times until the loop criteria is satisfied. This applies to all the Repeat string instruction prefixes (i.e., REP, REPE, REPZ, REPNE, and REPNZ). This counter will also only increment once per each HLT instruction executed regardless of how many cycles the processor remains in the HALT state.
17H	INSTRUCTIONS_ EXECUTED_ V PIPE	Number of instructions executed in the V_pipe. The event indicates the number of instructions that were paired.	This event is the same as the 16H event except it only counts the number of instructions actually executed in the V-pipe.
18H	BUS_CYCLE_ DURATION	Number of clocks while a bus cycle is in progress. This event measures bus use.	The count includes HLDA, AHOLD, and BOFF# clocks.
19H	WRITE_BUFFER_ FULL_STALL_ DURATION	Number of clocks while the pipeline is stalled due to full write buffers.	Full write buffers stall data memory read misses, data memory write misses, and data memory write hits to S-state lines. Stalls on I/O accesses are not included.

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
1AH	WAITING_FOR_ DATA_MEMORY_ READ_STALL_ DURATION	Number of clocks while the pipeline is stalled while waiting for data memory reads.	Data TLB Miss processing is also included in the count. The pipeline stalls while a data memory read is in progress including attempts to read that are not bypassed while a line is being filled.
1BH	STALL ON WRITE TO AN E- OR M-STATE LINE	Number of stalls on writes to E- or M-state lines.	
1CH	LOCKED BUS CYCLE	Number of locked bus cycles that occur as the result of the LOCK prefix or LOCK instruction, page-table updates, and descriptor table updates.	Only the read portion of the locked read-modify-write is counted. Split locked cycles (SCYC active) count as two separate accesses. Cycles restarted due to BOFF# are not re-counted.
1DH	I/O READ OR WRITE CYCLE	Number of bus cycles directed to I/O space.	Misaligned I/O accesses will generate two bus cycles. Bus cycles restarted due to BOFF# are not re-counted.
1EH	NONCACHEABLE_ MEMORY_READS	Number of noncacheable instruction or data memory read bus cycles. The count includes read cycles caused by TLB misses, but does not include read cycles to I/O space.	Cycles restarted due to BOFF# are not re-counted.
1FH	PIPELINE_AGI_ STALLS	Number of address generation interlock (AGI) stalls. An AGI occurring in both the U- and V-pipelines in the same clock signals this event twice.	An AGI occurs when the instruction in the execute stage of either of U- or V-pipelines is writing to either the index or base address register of an instruction in the D2 (address generation) stage of either the U- or V- pipelines.
20H	Reserved		
21H	Reserved		
22H	FLOPS	Number of floating-point operations that occur.	Number of floating-point adds, subtracts, multiplies, divides, remainders, and square roots are counted. The transcendental instructions consist of multiple adds and multiplies and will signal this event multiple times. Instructions generating the divide-by-zero, negative square root, special operand, or stack exceptions will not be counted. Instructions generating all other floating-point exceptions will be counted. The integer multiply instructions and other instructions which use the x87 FPU will be counted.
23H	BREAKPOINT MATCH ON DR0 REGISTER	Number of matches on register DR0 breakpoint.	The counters is incremented regardless if the breakpoints are enabled or not. However, if breakpoints are not enabled, code breakpoint matches will not be checked for instructions executed in the V-pipe and will not cause this counter to be incremented. (They are checked on instruction executed in the U-pipe only when breakpoints are not enabled.) These events correspond to the signals driven on the BP[3:0] pins. Refer to Chapter 17, “Debug, Branch Profile, TSC, and Resource Monitoring Features” for more information.
24H	BREAKPOINT MATCH ON DR1 REGISTER	Number of matches on register DR1 breakpoint.	See comment for 23H event.

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
25H	BREAKPOINT MATCH ON DR2 REGISTER	Number of matches on register DR2 breakpoint.	See comment for 23H event.
26H	BREAKPOINT MATCH ON DR3 REGISTER	Number of matches on register DR3 breakpoint.	See comment for 23H event.
27H	HARDWARE INTERRUPTS	Number of taken INTR and NMI interrupts.	
28H	DATA_READ_OR_ WRITE	Number of memory data reads and/or writes (internal data cache hit and miss combined).	Split cycle reads and writes are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
29H	DATA_READ_MISS OR_WRITE MISS	Number of memory read and/or write accesses that miss the internal data cache, whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
2AH	BUS_OWNERSHIP_ LATENCY (Counter 0)	The time from LRM bus ownership request to bus ownership granted (that is, the time from the earlier of a PBREQ (0), PHITM# or HITM# assertion to a PBGNT assertion)	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2AH	BUS OWNERSHIP TRANSFERS (Counter 1)	The number of buss ownership transfers (that is, the number of PBREQ (0) assertions	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ U-PIPE (Counter 0)	Number of MMX instructions executed in the U-pipe	
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ V-PIPE (Counter 1)	Number of MMX instructions executed in the V-pipe	
2CH	CACHE_M-STATE_LINE_ SHARING (Counter 0)	Number of times a processor identified a hit to a modified line due to a memory access in the other processor (PHITM (O))	If the average memory latencies of the system are known, this event enables the user to count the Write Backs on PHITM(O) penalty and the Latency on Hit Modified(I) penalty.
2CH	CACHE_LINE_ SHARING (Counter 1)	Number of shared data lines in the L1 cache (PHIT (O))	
2DH	EMMS_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of EMMS instructions executed	

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
2DH	TRANSITIONS_ BETWEEN_MMX_ AND_FP_ INSTRUCTIONS (Counter 1)	Number of transitions between MMX and floating-point instructions or vice versa An even count indicates the processor is in MMX state. an odd count indicates it is in FP state.	This event counts the first floating-point instruction following an MMX instruction or first MMX instruction following a floating-point instruction. The count may be used to estimate the penalty in transitions between floating-point state and MMX state.
2EH	BUS_UTILIZATION_ DUE_TO_ PROCESSOR_ ACTIVITY (Counter 0)	Number of clocks the bus is busy due to the processor's own activity (the bus activity that is caused by the processor)	
2EH	WRITES_TO_ NONCACHEABLE_ MEMORY (Counter 1)	Number of write accesses to noncacheable memory	The count includes write cycles caused by TLB misses and I/O write cycles. Cycles restarted due to BOFF# are not re-counted.
2FH	SATURATING_ MMX_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of saturating MMX instructions executed, independently of whether they actually saturated.	
2FH	SATURATIONS_ PERFORMED (Counter 1)	Number of MMX instructions that used saturating arithmetic when at least one of its results actually saturated	If an MMX instruction operating on 4 doublewords saturated in three out of the four results, the counter will be incremented by one only.
30H	NUMBER_OF_ CYCLES_NOT_IN_ HALT_STATE (Counter 0)	Number of cycles the processor is not idle due to HLT instruction	This event will enable the user to calculate "net CPI". Note that during the time that the processor is executing the HLT instruction, the Time-Stamp Counter is not disabled. Since this event is controlled by the Counter Controls CC0, CC1 it can be used to calculate the CPI at CPL=3, which the TSC cannot provide.
30H	DATA_CACHE_ TLB_MISS_ STALL_DURATION (Counter 1)	Number of clocks the pipeline is stalled due to a data cache translation look-aside buffer (TLB) miss	
31H	MMX_ INSTRUCTION_ DATA_READS (Counter 0)	Number of MMX instruction data reads	
31H	MMX_ INSTRUCTION_ DATA_READ_ MISSES (Counter 1)	Number of MMX instruction data read misses	
32H	FLOATING_POINT_S TALLS_DURATION (Counter 0)	Number of clocks while pipe is stalled due to a floating-point freeze	
32H	TAKEN_BRANCHES (Counter 1)	Number of taken branches	

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
33H	D1_STARVATION_ AND_FIFO_IS_ EMPTY (Counter 0)	Number of times D1 stage cannot issue ANY instructions since the FIFO buffer is empty	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer.
33H	D1_STARVATION_ AND_ONLY_ONE_ INSTRUCTION_IN_ FIFO (Counter 1)	Number of times the D1 stage issues a single instruction (since the FIFO buffer had just one instruction ready)	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer. When combined with the previously defined events, Instruction Executed (16H) and Instruction Executed in the V-pipe (17H), this event enables the user to calculate the numbers of time pairing rules prevented issuing of two instructions.
34H	MMX_ INSTRUCTION_ DATA_WRITES (Counter 0)	Number of data writes caused by MMX instructions	
34H	MMX_ INSTRUCTION_ DATA_WRITE_ MISSES (Counter 1)	Number of data write misses caused by MMX instructions	
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS (Counter 0)	Number of pipeline flushes due to wrong branch predictions resolved in either the E-stage or the WB-stage	The count includes any pipeline flush due to a branch that the pipeline did not follow correctly. It includes cases where a branch was not in the BTB, cases where a branch was in the BTB but was mispredicted, and cases where a branch was correctly predicted but to the wrong address. Branches are resolved in either the Execute stage (E-stage) or the Writeback stage (WB-stage). In the later case, the misprediction penalty is larger by one clock. The difference between the 35H event count in counter 0 and counter 1 is the number of E-stage resolved branches.
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS_ RESOLVED_IN_ WB-STAGE (Counter 1)	Number of pipeline flushes due to wrong branch predictions resolved in the WB-stage	See note for event 35H (Counter 0).
36H	MISALIGNED_ DATA_MEMORY_ REFERENCE_ON_ MMX_ INSTRUCTIONS (Counter 0)	Number of misaligned data memory references when executing MMX instructions	
36H	PIPELINE_ ISTALL_FOR_MMX_ INSTRUCTION_ DATA_MEMORY_ READS (Counter 1)	Number clocks during pipeline stalls caused by waits form MMX instruction data memory reads	T3:

Table 19-39. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
37H	MISPREDICTED_ OR_ UNPREDICTED_ RETURNS (Counter 1)	Number of returns predicted incorrectly or not predicted at all	The count is the difference between the total number of executed returns and the number of returns that were correctly predicted. Only RET instructions are counted (for example, IRET instructions are not counted).
37H	PREDICTED_ RETURNS (Counter 1)	Number of predicted returns (whether they are predicted correctly and incorrectly	Only RET instructions are counted (for example, IRET instructions are not counted).
38H	MMX_MULTIPLY_ UNIT_INTERLOCK (Counter 0)	Number of clocks the pipe is stalled since the destination of previous MMX multiply instruction is not ready yet	The counter will not be incremented if there is another cause for a stall. For each occurrence of a multiply interlock, this event will be counted twice (if the stalled instruction comes on the next clock after the multiply) or by once (if the stalled instruction comes two clocks after the multiply).
38H	MOVD/MOVQ_ STORE_STALL_ DUE_TO_ PREVIOUS_MMX_ OPERATION (Counter 1)	Number of clocks a MOVD/MOVQ instruction store is stalled in D2 stage due to a previous MMX operation with a destination to be used in the store instruction.	
39H	RETURNS (Counter 0)	Number or returns executed.	Only RET instructions are counted; IRET instructions are not counted. Any exception taken on a RET instruction and any interrupt recognized by the processor on the instruction boundary prior to the execution of the RET instruction will also cause this counter to be incremented.
39H	Reserved		
3AH	BTB_FALSE_ ENTRIES (Counter 0)	Number of false entries in the Branch Target Buffer	False entries are causes for misprediction other than a wrong prediction.
3AH	BTB_MISS_ PREDICTION_ON_ NOT-TAKEN_ BRANCH (Counter 1)	Number of times the BTB predicted a not-taken branch as taken	
3BH	FULL_WRITE_ BUFFER_STALL_ DURATION_ WHILE_ EXECUTING_MMX_I NSTRUCTIONS (Counter 0)	Number of clocks while the pipeline is stalled due to full write buffers while executing MMX instructions	
3BH	STALL_ON_MMX_ INSTRUCTION_ WRITE_TO E-_OR_ M-STATE_LINE (Counter 1)	Number of clocks during stalls on MMX instructions writing to E- or M-state lines	

14. Updates to Chapter 22, Volume 3B

Change bars show changes to Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Updated term "IA-32e paging" to "4-level paging"; included footnote on first usage of new term. Updated titles of sections 22.17 "Stack Operations and User Software" and 22.31 "Stack Operations and Supervisor Software".

CHAPTER 22
ARCHITECTURE COMPATIBILITY

Intel 64 and IA-32 processors are binary compatible. Compatibility means that, within limited constraints, programs that execute on previous generations of processors will produce identical results when executed on later processors. The compatibility constraints and any implementation differences between the Intel 64 and IA-32 processors are described in this chapter.

Each new processor has enhanced the software visible architecture from that found in earlier Intel 64 and IA-32 processors. Those enhancements have been defined with consideration for compatibility with previous and future processors. This chapter also summarizes the compatibility considerations for those extensions.

22.1 PROCESSOR FAMILIES AND CATEGORIES

IA-32 processors are referred to in several different ways in this chapter, depending on the type of compatibility information being related, as described in the following:

- **IA-32 Processors** — All the Intel processors based on the Intel IA-32 Architecture, which include the 8086/88, Intel 286, Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **32-bit Processors** — All the IA-32 processors that use a 32-bit architecture, which include the Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **16-bit Processors** — All the IA-32 processors that use a 16-bit architecture, which include the 8086/88 and Intel 286 processors.
- **P6 Family Processors** — All the IA-32 processors that are based on the P6 microarchitecture, which include the Pentium Pro, Pentium II, and Pentium III processors.
- **Pentium® 4 Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst® microarchitecture.
- **Intel® Pentium® M Processors** — A family of IA-32 processors that are based on the Intel Pentium M processor microarchitecture.
- **Intel® Core™ Duo and Solo Processors** — Families of IA-32 processors that are based on an improved Intel Pentium M processor microarchitecture.
- **Intel® Xeon® Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst microarchitecture. This family includes the Intel Xeon processor and the Intel Xeon processor MP based on the Intel NetBurst microarchitecture. Intel Xeon processors 3000, 3100, 3200, 3300, 3200, 5100, 5200, 5300, 5400, 7200, 7300 series are based on Intel Core microarchitectures and support Intel 64 architecture.
- **Pentium® D Processors** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture.
- **Pentium® Processor Extreme Editions** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture and supports Intel Hyper-Threading Technology.
- **Intel® Core™ 2 Processor family**— A family of Intel 64 processors that are based on the Intel Core microarchitecture. Intel Pentium Dual-Core processors are also based on the Intel Core microarchitecture.
- **Intel® Atom™ Processors** — A family of IA-32 and Intel 64 processors. 45 nm Intel Atom processors are based on the Intel Atom microarchitecture. 32 nm Intel Atom processors are based on newer microarchitectures including the Silvermont microarchitecture and the Airmont microarchitecture. Each generation of Intel Atom processors can be identified by the CPUID's DisplayFamily_DisplayModel signature; see Table 2-1 "CPUID Signature Values of DisplayFamily_DisplayModel" in Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

22.2 RESERVED BITS

Throughout this manual, certain bits are marked as reserved in many register and memory layout descriptions. When bits are marked as undefined or reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown effect. Software should follow these guidelines in dealing with reserved bits:

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

Software written for existing IA-32 processor that handles reserved bits correctly will port to future IA-32 processors without generating protection exceptions.

22.3 ENABLING NEW FUNCTIONS AND MODES

Most of the new control functions defined for the P6 family and Pentium processors are enabled by new mode flags in the control registers (primarily register CR4). This register is undefined for IA-32 processors earlier than the Pentium processor. Attempting to access this register with an Intel486 or earlier IA-32 processor results in an invalid-opcode exception (#UD). Consequently, programs that execute correctly on the Intel486 or earlier IA-32 processor cannot erroneously enable these functions. Attempting to set a reserved bit in register CR4 to a value other than its original value results in a general-protection exception (#GP). So, programs that execute on the P6 family and Pentium processors cannot erroneously enable functions that may be implemented in future IA-32 processors.

The P6 family and Pentium processors do not check for attempts to set reserved bits in model-specific registers; however these bits may be checked on more recent processors. It is the obligation of the software writer to enforce this discipline. These reserved bits may be used in future Intel processors.

22.4 DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE

Software can check for the presence of new architectural features and extensions in either of two ways:

1. Test for the presence of the feature or extension. Software can test for the presence of new flags in the EFLAGS register and control registers. If these flags are reserved (meaning not present in the processor executing the test), an exception is generated. Likewise, software can attempt to execute a new instruction, which results in an invalid-opcode exception (#UD) being generated if it is not supported.
2. Execute the CPUID instruction. The CPUID instruction (added to the IA-32 in the Pentium processor) indicates the presence of new features directly.

See Chapter 19, "Processor Identification and Feature Determination," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on detecting new processor features and extensions.

22.5 INTEL MMX TECHNOLOGY

The Pentium processor with MMX technology introduced the MMX technology and a set of MMX instructions to the IA-32. The MMX instructions are described in Chapter 9, "Programming with Intel® MMX™ Technology," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*. The MMX technology and MMX instructions are also included in the Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.

22.6 STREAMING SIMD EXTENSIONS (SSE)

The Streaming SIMD Extensions (SSE) were introduced in the Pentium III processor. The SSE extensions consist of a new set of instructions and a new set of registers. The new registers include the eight 128-bit XMM registers and the 32-bit MXCSR control and status register. These instructions and registers are designed to allow SIMD computations to be made on single-precision floating-point numbers. Several of these new instructions also operate in the MMX registers. SSE instructions and registers are described in Section 10, “Programming with Streaming SIMD Extensions (SSE),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

22.7 STREAMING SIMD EXTENSIONS 2 (SSE2)

The Streaming SIMD Extensions 2 (SSE2) were introduced in the Pentium 4 and Intel Xeon processors. They consist of a new set of instructions that operate on the XMM and MXCSR registers and perform SIMD operations on double-precision floating-point values and on integer values. Several of these new instructions also operate in the MMX registers. SSE2 instructions and registers are described in Chapter 11, “Programming with Streaming SIMD Extensions 2 (SSE2),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

22.8 STREAMING SIMD EXTENSIONS 3 (SSE3)

The Streaming SIMD Extensions 3 (SSE3) were introduced in Pentium 4 processors supporting Intel Hyper-Threading Technology and Intel Xeon processors. SSE3 extensions include 13 instructions. Ten of these 13 instructions support the single instruction multiple data (SIMD) execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion to integer. The remaining two instructions (MONITOR and MWAIT) accelerate synchronization of threads. SSE3 instructions are described in Chapter 12, “Programming with SSE3, SSSE3 and SSE4,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

22.9 ADDITIONAL STREAMING SIMD EXTENSIONS

The Supplemental Streaming SIMD Extensions 3 (SSSE3) were introduced in the Intel Core 2 processor and Intel Xeon processor 5100 series. Streaming SIMD Extensions 4 provided 54 new instructions introduced in 45 nm Intel Xeon processors and Intel Core 2 processors. SSSE3, SSE4.1 and SSE4.2 instructions are described in Chapter 12, “Programming with SSE3, SSSE3 and SSE4,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*.

22.10 INTEL HYPER-THREADING TECHNOLOGY

Intel Hyper-Threading Technology provides two logical processors that can execute two separate code streams (called *threads*) concurrently by using shared resources in a single processor core or in a physical package.

This feature was introduced in the Intel Xeon processor MP and later steppings of the Intel Xeon processor, and Pentium 4 processors supporting Intel Hyper-Threading Technology. The feature is also found in the Pentium processor Extreme Edition. See also: Section 8.7, “Intel® Hyper-Threading Technology Architecture.”

45 nm and 32 nm Intel Atom processors support Intel Hyper-Threading Technology.

Intel Atom processors based on Silvermont and Airmont microarchitectures do not support Intel Hyper-Threading Technology

22.11 MULTI-CORE TECHNOLOGY

The Pentium D processor and Pentium processor Extreme Edition provide two processor cores in each physical processor package. See also: Section 8.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology,” and Section 8.8, “Multi-Core Architecture.” Intel Core 2 Duo, Intel Pentium Dual-Core processors, Intel Xeon processors 3000, 3100, 5100, 5200 series provide two processor cores in each physical processor package. Intel Core 2 Extreme, Intel Core 2 Quad processors, Intel Xeon processors 3200, 3300, 5300, 5400, 7300 series provide two processor cores in each physical processor package.

22.12 SPECIFIC FEATURES OF DUAL-CORE PROCESSOR

Dual-core processors may have some processor-specific features. Use CPUID feature flags to detect the availability features. Note the following:

- **CPUID Brand String** — On Pentium processor Extreme Edition, the process will report the correct brand string only after the correct microcode updates are loaded.
- **Enhanced Intel SpeedStep Technology** — This feature is supported in Pentium D processor but not in Pentium processor Extreme Edition.

22.13 NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS

Table 22-1 identifies the instructions introduced into the IA-32 in the Pentium processor and later IA-32 processors.

22.13.1 Instructions Added Prior to the Pentium Processor

The following instructions were added in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLPG (invalidate TLB entry) instruction.

Table 22-1. New Instruction in the Pentium Processor and Later IA-32 Processors

Instruction	CPUID Identification Bits	Introduced In
CMOV*cc* (conditional move)	EDX, Bit 15	Pentium Pro processor
FCMOV*cc* (floating-point conditional move)	EDX, Bits 0 and 15	
FCOMI (floating-point compare and set EFLAGS)	EDX, Bits 0 and 15	
RDPMC (read performance monitoring counters)	EAX, Bits 8-11, set to 6H; see Note 1	
UD2 (undefined)	EAX, Bits 8-11, set to 6H	
CMPXCHG8B (compare and exchange 8 bytes)	EDX, Bit 8	Pentium processor
CPUID (CPU identification)	None; see Note 2	
RDTSC (read time-stamp counter)	EDX, Bit 4	
RDMSR (read model-specific register)	EDX, Bit 5	
WRMSR (write model-specific register)	EDX, Bit 5	
MMX Instructions	EDX, Bit 23	

Table 22-1. New Instruction in the Pentium Processor and Later IA-32 Processors (Contd.)

Instruction	CPUID Identification Bits	Introduced In

NOTES:

1. The RDPMC instruction was introduced in the P6 family of processors and added to later model Pentium processors. This instruction is model specific in nature and not architectural.
2. The CPUID instruction is available in all Pentium and P6 family processors and in later models of the Intel486 processors. The ability to set and clear the ID flag (bit 21) in the EFLAGS register indicates the availability of the CPUID instruction.

The following instructions were added in the Intel386 processor:

- LSS, LFS, and LGS (load SS, FS, and GS registers).
- Long-displacement conditional jumps.
- Single-bit instructions.
- Bit scan instructions.
- Double-shift instructions.
- Byte set on condition instruction.
- Move with sign/zero extension.
- Generalized multiply instruction.
- MOV to and from control registers.
- MOV to and from test registers (now obsolete).
- MOV to and from debug registers.
- RSM (resume from SMM). This instruction was introduced in the Intel386 SL and Intel486 SL processors.

The following instructions were added in the Intel 387 math coprocessor:

- FPREM1.
- FUCOM, FUCOMP, and FUCOMPP.

22.14 OBSOLETE INSTRUCTIONS

The MOV to and from test registers instructions were removed from the Pentium processor and future IA-32 processors. Execution of these instructions generates an invalid-opcode exception (#UD).

22.15 UNDEFINED OPCODES

All new instructions defined for IA-32 processors use binary encodings that were reserved on earlier-generation processors. Attempting to execute a reserved opcode always results in an invalid-opcode (#UD) exception being generated. Consequently, programs that execute correctly on earlier-generation processors cannot erroneously execute these instructions and thereby produce unexpected results when executed on later IA-32 processors.

22.16 NEW FLAGS IN THE EFLAGS REGISTER

The section titled “EFLAGS Register” in Chapter 3, “Basic Execution Environment,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the configuration of flags in the EFLAGS register for the P6 family processors. No new flags have been added to this register in the P6 family processors. The flags added to this register in the Pentium and Intel486 processors are described in the following sections.

The following flags were added to the EFLAGS register in the Pentium processor:

- VIF (virtual interrupt flag), bit 19.
- VIP (virtual interrupt pending), bit 20.

- ID (identification flag), bit 21.

The AC flag (bit 18) was added to the EFLAGS register in the Intel486 processor.

22.16.1 Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors

The following bits in the EFLAGS register that can be used to differentiate between the 32-bit IA-32 processors:

- Bit 18 (the AC flag) can be used to distinguish an Intel386 processor from the P6 family, Pentium, and Intel486 processors. Since it is not implemented on the Intel386 processor, it will always be clear.
- Bit 21 (the ID flag) indicates whether an application can execute the CPUID instruction. The ability to set and clear this bit indicates that the processor is a P6 family or Pentium processor. The CPUID instruction can then be used to determine which processor.
- Bits 19 (the VIF flag) and 20 (the VIP flag) will always be zero on processors that do not support virtual mode extensions, which includes all 32-bit processors prior to the Pentium processor.

See Chapter 19, "Processor Identification and Feature Determination," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on identifying processors.

22.17 STACK OPERATIONS AND USER SOFTWARE

This section identifies the differences in stack implementation between the various IA-32 processors.

22.17.1 PUSH SP

The P6 family, Pentium, Intel486, Intel386, and Intel 286 processors push a different value on the stack for a PUSH SP instruction than the 8086 processor. The 32-bit processors push the value of the SP register before it is decremented as part of the push operation; the 8086 processor pushes the value of the SP register after it is decremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV  BP, SP
XCHG BP, [BP]
```

This code functions as the 8086 processor PUSH SP instruction on the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors.

22.17.2 EFLAGS Pushed on the Stack

The setting of the stored values of bits 12 through 15 (which includes the IOPL field and the NT flag) in the EFLAGS register by the PUSHF instruction, by interrupts, and by exceptions is different with the 32-bit IA-32 processors than with the 8086 and Intel 286 processors. The differences are as follows:

- 8086 processor—bits 12 through 15 are always set.
- Intel 286 processor—bits 12 through 15 are always cleared in real-address mode.
- 32-bit processors in real-address mode—bit 15 (reserved) is always cleared, and bits 12 through 14 have the last value loaded into them.

22.18 X87 FPU

This section addresses the issues that must be faced when porting floating-point software designed to run on earlier IA-32 processors and math coprocessors to a Pentium 4, Intel Xeon, P6 family, or Pentium processor with integrated x87 FPU. To software, a Pentium 4, Intel Xeon, or P6 family processor looks very much like a Pentium processor. Floating-point software which runs on a Pentium or Intel486 DX processor, or on an Intel486 SX

processor/Intel 487 SX math coprocessor system or an Intel386 processor/Intel 387 math coprocessor system, will run with at most minor modifications on a Pentium 4, Intel Xeon, or P6 family processor. To port code directly from an Intel 286 processor/Intel 287 math coprocessor system or an Intel 8086 processor/8087 math coprocessor system to a Pentium 4, Intel Xeon, P6 family, or Pentium processor, certain additional issues must be addressed.

In the following sections, the term "32-bit x87 FPUs" refers to the P6 family, Pentium, and Intel486 DX processors, and to the Intel 487 SX and Intel 387 math coprocessors; the term "16-bit IA-32 math coprocessors" refers to the Intel 287 and 8087 math coprocessors.

22.18.1 Control Register CR0 Flags

The ET, NE, and MP flags in control register CR0 control the interface between the integer unit of an IA-32 processor and either its internal x87 FPU or an external math coprocessor. The effect of these flags in the various IA-32 processors are described in the following paragraphs.

The ET (extension type) flag (bit 4 of the CR0 register) is used in the Intel386 processor to indicate whether the math coprocessor in the system is an Intel 287 math coprocessor (flag is clear) or an Intel 387 DX math coprocessor (flag is set). This bit is hardwired to 1 in the P6 family, Pentium, and Intel486 processors.

The NE (Numeric Exception) flag (bit 5 of the CR0 register) is used in the P6 family, Pentium, and Intel486 processors to determine whether unmasked floating-point exceptions are reported internally through interrupt vector 16 (flag is set) or externally through an external interrupt (flag is clear). On a hardware reset, the NE flag is initialized to 0, so software using the automatic internal error-reporting mechanism must set this flag to 1. This flag is nonexistent on the Intel386 processor.

As on the Intel 286 and Intel386 processors, the MP (monitor coprocessor) flag (bit 1 of register CR0) determines whether the WAIT/FWAIT instructions or waiting-type floating-point instructions trap when the context of the x87 FPU is different from that of the currently-executing task. If the MP and TS flag are set, then a WAIT/FWAIT instruction and waiting instructions will cause a device-not-available exception (interrupt vector 7). The MP flag is used on the Intel 286 and Intel386 processors to support the use of a WAIT/FWAIT instruction to wait on a device other than a math coprocessor. The device reports its status through the BUSY# pin. Since the P6 family, Pentium, and Intel486 processors do not have such a pin, the MP flag has no relevant use and should be set to 1 for normal operation.

22.18.2 x87 FPU Status Word

This section identifies differences to the x87 FPU status word for the different IA-32 processors and math coprocessors, the reason for the differences, and their impact on software.

22.18.2.1 Condition Code Flags (C0 through C3)

The following information pertains to differences in the use of the condition code flags (C0 through C3) located in bits 8, 9, 10, and 14 of the x87 FPU status word.

After execution of an FINIT instruction or a hardware reset on a 32-bit x87 FPU, the condition code flags are set to 0. The same operations on a 16-bit IA-32 math coprocessor leave these flags intact (they contain their prior value). This difference in operation has no impact on software and provides a consistent state after reset.

Transcendental instruction results in the core range of the P6 family and Pentium processors may differ from the Intel486 DX processor and Intel 487 SX math coprocessor by 2 to 3 units in the last place (ulps)—(see "Transcendental Instruction Accuracy" in Chapter 8, "Programming with the x87 FPU," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). As a result, the value saved in the C1 flag may also differ.

After an incomplete FPREM/FPREM1 instruction, the C0, C1, and C3 flags are set to 0 on the 32-bit x87 FPUs. After the same operation on a 16-bit IA-32 math coprocessor, these flags are left intact.

On the 32-bit x87 FPUs, the C2 flag serves as an incomplete flag for the FTAN instruction. On the 16-bit IA-32 math coprocessors, the C2 flag is undefined for the FPTAN instruction. This difference has no impact on software, because Intel 287 or 8087 programs do not check C2 after an FPTAN instruction. The use of this flag on later processors allows fast checking of operand range.

22.18.2.2 Stack Fault Flag

When unmasked stack overflow or underflow occurs on a 32-bit x87 FPU, the IE flag (bit 0) and the SF flag (bit 6) of the x87 FPU status word are set to indicate a stack fault and condition code flag C1 is set or cleared to indicate overflow or underflow, respectively. When unmasked stack overflow or underflow occurs on a 16-bit IA-32 math coprocessor, only the IE flag is set. Bit 6 is reserved on these processors. The addition of the SF flag on a 32-bit x87 FPU has no impact on software. Existing exception handlers need not change, but may be upgraded to take advantage of the additional information.

22.18.3 x87 FPU Control Word

Only affine closure is supported for infinity control on a 32-bit x87 FPU. The infinity control flag (bit 12 of the x87 FPU control word) remains programmable on these processors, but has no effect. This change was made to conform to the IEEE Standard 754 for Binary Floating-Point Arithmetic. On a 16-bit IA-32 math coprocessor, both affine and projective closures are supported, as determined by the setting of bit 12. After a hardware reset, the default value of bit 12 is projective. Software that requires projective infinity arithmetic may give different results.

22.18.4 x87 FPU Tag Word

When loading the tag word of a 32-bit x87 FPU, using an FLDENV, FRSTOR, or FXRSTOR (Pentium III processor only) instruction, the processor examines the incoming tag and classifies the location only as empty or non-empty. Thus, tag values of 00, 01, and 10 are interpreted by the processor to indicate a non-empty location. The tag value of 11 is interpreted by the processor to indicate an empty location. Subsequent operations on a non-empty register always examine the value in the register, not the value in its tag. The FSTENV, FSAVE, and FXSAVE (Pentium III processor only) instructions examine the non-empty registers and put the correct values in the tags before storing the tag word.

The corresponding tag for a 16-bit IA-32 math coprocessor is checked before each register access to determine the class of operand in the register; the tag is updated after every change to a register so that the tag always reflects the most recent status of the register. Software can load a tag with a value that disagrees with the contents of a register (for example, the register contains a valid value, but the tag says special). Here, the 16-bit IA-32 math coprocessors honor the tag and do not examine the register.

Software written to run on a 16-bit IA-32 math coprocessor may not operate correctly on a 16-bit x87 FPU, if it uses the FLDENV, FRSTOR, or FXRSTOR instructions to change tags to values (other than to empty) that are different from actual register contents.

The encoding in the tag word for the 32-bit x87 FPUs for unsupported data formats (including pseudo-zero and unnormal) is special (10B), to comply with IEEE Standard 754. The encoding in the 16-bit IA-32 math coprocessors for pseudo-zero and unnormal is valid (00B) and the encoding for other unsupported data formats is special (10B). Code that recognizes the pseudo-zero or unnormal format as valid must therefore be changed if it is ported to a 32-bit x87 FPU.

22.18.5 Data Types

This section discusses the differences of data types for the various x87 FPUs and math coprocessors.

22.18.5.1 NaNs

The 32-bit x87 FPUs distinguish between signaling NaNs (SNaNs) and quiet NaNs (QNaNs). These x87 FPUs only generate QNaNs and normally do not generate an exception upon encountering a QNaN. An invalid-operation exception (#I) is generated only upon encountering a SNaN, except for the FCOM, FIST, and FBSTP instructions, which also generates an invalid-operation exceptions for a QNaNs. This behavior matches IEEE Standard 754.

The 16-bit IA-32 math coprocessors only generate one kind of NaN (the equivalent of a QNaN), but the raise an invalid-operation exception upon encountering any kind of NaN.

When porting software written to run on a 16-bit IA-32 math coprocessor to a 32-bit x87 FPU, uninitialized memory locations that contain QNaNs should be changed to SNaNs to cause the x87 FPU or math coprocessor to fault when uninitialized memory locations are referenced.

22.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats

The 32-bit x87 FPUs neither generate nor support the pseudo-zero, pseudo-NaN, pseudo-infinity, and unnormal formats. Whenever they encounter them in an arithmetic operation, they raise an invalid-operation exception. The 16-bit IA-32 math coprocessors define and support special handling for these formats. Support for these formats was dropped to conform with IEEE Standard 754 for Binary Floating-Point Arithmetic.

This change should not impact software ported from 16-bit IA-32 math coprocessors to 32-bit x87 FPUs. The 32-bit x87 FPUs do not generate these formats, and therefore will not encounter them unless software explicitly loads them in the data registers. The only affect may be in how software handles the tags in the tag word (see also: Section 22.18.4, “x87 FPU Tag Word”).

22.18.6 Floating-Point Exceptions

This section identifies the implementation differences in exception handling for floating-point instructions in the various x87 FPUs and math coprocessors.

22.18.6.1 Denormal Operand Exception (#D)

When the denormal operand exception is masked, the 32-bit x87 FPUs automatically normalize denormalized numbers when possible; whereas, the 16-bit IA-32 math coprocessors return a denormal result. A program written to run on a 16-bit IA-32 math coprocessor that uses the denormal exception solely to normalize denormalized operands is redundant when run on the 32-bit x87 FPUs. If such a program is run on 32-bit x87 FPUs, performance can be improved by masking the denormal exception. Floating-point programs run faster when the FPU performs normalization of denormalized operands.

The denormal operand exception is not raised for transcendental instructions and the FXTRACT instruction on the 16-bit IA-32 math coprocessors. This exception is raised for these instructions on the 32-bit x87 FPUs. The exception handlers ported to these latter processors need to be changed only if the handlers gives special treatment to different opcodes.

22.18.6.2 Numeric Overflow Exception (#O)

On the 32-bit x87 FPUs, when the numeric overflow exception is masked and the rounding mode is set to chop (toward 0), the result is the largest positive or smallest negative number. The 16-bit IA-32 math coprocessors do not signal the overflow exception when the masked response is not ∞; that is, they signal overflow only when the rounding control is not set to round to 0. If rounding is set to chop (toward 0), the result is positive or negative ∞. Under the most common rounding modes, this difference has no impact on existing software.

If rounding is toward 0 (chop), a program on a 32-bit x87 FPU produces, under overflow conditions, a result that is different in the least significant bit of the significand, compared to the result on a 16-bit IA-32 math coprocessor. The reason for this difference is IEEE Standard 754 compatibility.

When the overflow exception is not masked, the precision exception is flagged on the 32-bit x87 FPUs. When the result is stored in the stack, the significand is rounded according to the precision control (PC) field of the FPU control word or according to the opcode. On the 16-bit IA-32 math coprocessors, the precision exception is not flagged and the significand is not rounded. The impact on existing software is that if the result is stored on the stack, a program running on a 32-bit x87 FPU produces a different result under overflow conditions than on a 16-bit IA-32 math coprocessor. The difference is apparent only to the exception handler. This difference is for IEEE Standard 754 compatibility.

22.18.6.3 Numeric Underflow Exception (#U)

When the underflow exception is masked on the 32-bit x87 FPUs, the underflow exception is signaled when the result is tiny and inexact (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). When the underflow exception is unmasked and the instruction is supposed to store the result on the stack, the significand is rounded to the appropriate precision (according to the PC flag in the FPU control word, for those instructions controlled by PC, otherwise to extended precision), after adjusting the exponent.

22.18.6.4 Exception Precedence

There is no difference in the precedence of the denormal-operand exception on the 32-bit x87 FPUs, whether it be masked or not. When the denormal-operand exception is not masked on the 16-bit IA-32 math coprocessors, it takes precedence over all other exceptions. This difference causes no impact on existing software, but some unneeded normalization of denormalized operands is prevented on the Intel486 processor and Intel 387 math coprocessor.

22.18.6.5 CS and EIP For FPU Exceptions

On the Intel 32-bit x87 FPUs, the values from the CS and EIP registers saved for floating-point exceptions point to any prefixes that come before the floating-point instruction. On the 8087 math coprocessor, the saved CS and IP registers points to the floating-point instruction.

22.18.6.6 FPU Error Signals

The floating-point error signals to the P6 family, Pentium, and Intel486 processors do not pass through an interrupt controller; an INT# signal from an Intel 387, Intel 287 or 8087 math coprocessors does. If an 8086 processor uses another exception for the 8087 interrupt, both exception vectors should call the floating-point-error exception handler. Some instructions in a floating-point-error exception handler may need to be deleted if they use the interrupt controller. The P6 family, Pentium, and Intel486 processors have signals that, with the addition of external logic, support reporting for emulation of the interrupt mechanism used in many personal computers.

On the P6 family, Pentium, and Intel486 processors, an undefined floating-point opcode will cause an invalid-opcode exception (#UD, interrupt vector 6). Undefined floating-point opcodes, like legal floating-point opcodes, cause a device not available exception (#NM, interrupt vector 7) when either the TS or EM flag in control register CR0 is set. The P6 family, Pentium, and Intel486 processors do not check for floating-point error conditions on encountering an undefined floating-point opcode.

22.18.6.7 Assertion of the FERR# Pin

When using the MS-DOS compatibility mode for handing floating-point exceptions, the FERR# pin must be connected to an input to an external interrupt controller. An external interrupt is then generated when the FERR# output drives the input to the interrupt controller and the interrupt controller in turn drives the INTR pin on the processor.

For the P6 family and Intel386 processors, an unmasked floating-point exception always causes the FERR# pin to be asserted upon completion of the instruction that caused the exception. For the Pentium and Intel486 processors, an unmasked floating-point exception may cause the FERR# pin to be asserted either at the end of the instruction causing the exception or immediately before execution of the next floating-point instruction. (Note that the next floating-point instruction would not be executed until the pending unmasked exception has been handled.) See Appendix D, "Guidelines for Writing x87 FPU Extension Handlers," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a complete description of the required mechanism for handling floating-point exceptions using the MS-DOS compatibility mode.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems; this approach also limits newer processors to operate with one logical processor active.

22.18.6.8 Invalid Operation Exception On Denormals

An invalid-operation exception is not generated on the 32-bit x87 FPUs upon encountering a denormal value when executing a FSQRT, FDIV, or FPREM instruction or upon conversion to BCD or to integer. The operation proceeds by first normalizing the value. On the 16-bit IA-32 math coprocessors, upon encountering this situation, the invalid-operation exception is generated. This difference has no impact on existing software. Software running on the 32-bit x87 FPUs continues to execute in cases where the 16-bit IA-32 math coprocessors trap. The reason for this change was to eliminate an exception from being raised.

22.18.6.9 Alignment Check Exceptions (#AC)

If alignment checking is enabled, a misaligned data operand on the P6 family, Pentium, and Intel486 processors causes an alignment check exception (#AC) when a program or procedure is running at privilege-level 3, except for the stack portion of the FSAVE/FNSAVE, FXSAVE, FRSTOR, and FXRSTOR instructions.

22.18.6.10 Segment Not Present Exception During FLDENV

On the Intel486 processor, when a segment not present exception (#NP) occurs in the middle of an FLDENV instruction, it can happen that part of the environment is loaded and part not. In such cases, the FPU control word is left with a value of 007FH. The P6 family and Pentium processors ensure the internal state is correct at all times by attempting to read the first and last bytes of the environment before updating the internal state.

22.18.6.11 Device Not Available Exception (#NM)

The device-not-available exception (#NM, interrupt 7) will occur in the P6 family, Pentium, and Intel486 processors as described in Section 2.5, "Control Registers," Table 2-2, and Chapter 6, "Interrupt 7—Device Not Available Exception (#NM)."

22.18.6.12 Coprocessor Segment Overrun Exception

The coprocessor segment overrun exception (interrupt 9) does not occur in the P6 family, Pentium, and Intel486 processors. In situations where the Intel 387 math coprocessor would cause an interrupt 9, the P6 family, Pentium, and Intel486 processors simply abort the instruction. To avoid undetected segment overruns, it is recommended that the floating-point save area be placed in the same page as the TSS. This placement will prevent the FPU environment from being lost if a page fault occurs during the execution of an FLDENV, FRSTOR, or FXRSTOR instruction while the operating system is performing a task switch.

22.18.6.13 General Protection Exception (#GP)

A general-protection exception (#GP, interrupt 13) occurs if the starting address of a floating-point operand falls outside a segment's size. An exception handler should be included to report these programming errors.

22.18.6.14 Floating-Point Error Exception (#MF)

In real mode and protected mode (not including virtual-8086 mode), interrupt vector 16 must point to the floating-point exception handler. In virtual-8086 mode, the virtual-8086 monitor can be programmed to accommodate a different location of the interrupt vector for floating-point exceptions.

22.18.7 Changes to Floating-Point Instructions

This section identifies the differences in floating-point instructions for the various Intel FPU and math coprocessor architectures, the reason for the differences, and their impact on software.

22.18.7.1 FDIV, FPREM, and FSQRT Instructions

The 32-bit x87 FPUs support operations on denormalized operands and, when detected, an underflow exception can occur, for compatibility with the IEEE Standard 754. The 16-bit IA-32 math coprocessors do not operate on denormalized operands or return underflow results. Instead, they generate an invalid-operation exception when they detect an underflow condition. An existing underflow exception handler will require change only if it gives different treatment to different opcodes. Also, it is possible that fewer invalid-operation exceptions will occur.

22.18.7.2 FSCALE Instruction

With the 32-bit x87 FPUs, the range of the scaling operand is not restricted. If (0 < | ST(1) < 1), the scaling factor is 0; therefore, ST(0) remains unchanged. If the rounded result is not exact or if there was a loss of accuracy (masked underflow), the precision exception is signaled. With the 16-bit IA-32 math coprocessors, the range of the scaling operand is restricted. If (0 < | ST(1) | < 1), the result is undefined and no exception is signaled. The impact of this difference on exiting software is that different results are delivered on the 32-bit and 16-bit FPUs and math coprocessors when (0 < | ST(1) | < 1).

22.18.7.3 FPREM1 Instruction

The 32-bit x87 FPUs compute a partial remainder according to IEEE Standard 754. This instruction does not exist on the 16-bit IA-32 math coprocessors. The availability of the FPREM1 instruction has is no impact on existing software.

22.18.7.4 FPREM Instruction

On the 32-bit x87 FPUs, the condition code flags C0, C3, C1 in the status word correctly reflect the three low-order bits of the quotient following execution of the FPREM instruction. On the 16-bit IA-32 math coprocessors, the quotient bits are incorrect when performing a reduction of $(64^N + M)$ when $(N \geq 1)$ and M is 1 or 2. This difference does not affect existing software; software that works around the bug should not be affected.

22.18.7.5 FUCOM, FUCOMP, and FUCOMPP Instructions

When executing the FUCOM, FUCOMP, and FUCOMPP instructions, the 32-bit x87 FPUs perform unordered compare according to IEEE Standard 754. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these new instructions has no impact on existing software.

22.18.7.6 FPTAN Instruction

On the 32-bit x87 FPUs, the range of the operand for the FPTAN instruction is much less restricted $(| ST(0) | < 2^{63})$ than on earlier math coprocessors. The instruction reduces the operand internally using an internal $\pi/4$ constant that is more accurate. The range of the operand is restricted to $(| ST(0) | < \pi/4)$ on the 16-bit IA-32 math coprocessors; the operand must be reduced to this range using FPREM. This change has no impact on existing software. See also sections 8.3.8 and section 8.3.10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for more information on the accuracy of the FPTAN instruction.

22.18.7.7 Stack Overflow

On the 32-bit x87 FPUs, if an FPU stack overflow occurs when the invalid-operation exception is masked, the FPU returns the real, integer, or BCD-integer indefinite value to the destination operand, depending on the instruction being executed. On the 16-bit IA-32 math coprocessors, the original operand remains unchanged following a stack overflow, but it is loaded into register ST(1). This difference has no impact on existing software.

22.18.7.8 FSIN, FCOS, and FSINCOS Instructions

On the 32-bit x87 FPUs, these instructions perform three common trigonometric functions. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these instructions has no impact on existing

software, but using them provides a performance upgrade. See also sections 8.3.8 and section 8.3.10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for more information on the accuracy of the FSIN, FCOS, and FSINCOS instructions.

22.18.7.9 FPATAN Instruction

On the 32-bit x87 FPUs, the range of operands for the FPATAN instruction is unrestricted. On the 16-bit IA-32 math coprocessors, the absolute value of the operand in register ST(0) must be smaller than the absolute value of the operand in register ST(1). This difference has impact on existing software.

22.18.7.10 F2XM1 Instruction

The 32-bit x87 FPUs support a wider range of operands ($-1 < ST(0) < +1$) for the F2XM1 instruction. The supported operand range for the 16-bit IA-32 math coprocessors is ($0 \le ST(0) \le 0.5$). This difference has no impact on existing software.

22.18.7.11 FLD Instruction

On the 32-bit x87 FPUs, when using the FLD instruction to load an extended-real value, a denormal-operand exception is not generated because the instruction is not arithmetic. The 16-bit IA-32 math coprocessors do report a denormal-operand exception in this situation. This difference does not affect existing software.

On the 32-bit x87 FPUs, loading a denormal value that is in single- or double-real format causes the value to be converted to extended-real format. Loading a denormal value on the 16-bit IA-32 math coprocessors causes the value to be converted to an unnormal. If the next instruction is FXTRACT or FXAM, the 32-bit x87 FPUs will give a different result than the 16-bit IA-32 math coprocessors. This change was made for IEEE Standard 754 compatibility.

On the 32-bit x87 FPUs, loading an SNaN that is in single- or double-real format causes the FPU to generate an invalid-operation exception. The 16-bit IA-32 math coprocessors do not raise an exception when loading a signaling NaN. The invalid-operation exception handler for 16-bit math coprocessor software needs to be updated to handle this condition when porting software to 32-bit FPUs. This change was made for IEEE Standard 754 compatibility.

22.18.7.12 FXTRACT Instruction

On the 32-bit x87 FPUs, if the operand is 0 for the FXTRACT instruction, the divide-by-zero exception is reported and $-\infty$ is delivered to register ST(1). If the operand is $+\infty$, no exception is reported. If the operand is 0 on the 16-bit IA-32 math coprocessors, 0 is delivered to register ST(1) and no exception is reported. If the operand is $+\infty$, the invalid-operation exception is reported. These differences have no impact on existing software. Software usually bypasses 0 and ∞. This change is due to the IEEE Standard 754 recommendation to fully support the "logb" function.

22.18.7.13 Load Constant Instructions

On 32-bit x87 FPUs, rounding control is in effect for the load constant instructions. Rounding control is not in effect for the 16-bit IA-32 math coprocessors. Results for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions are the same as for the 16-bit IA-32 math coprocessors when rounding control is set to round to nearest or round to $+\infty$. They are the same for the FLDL2T instruction when rounding control is set to round to nearest, round to $-\infty$, or round to zero. Results are different from the 16-bit IA-32 math coprocessors in the least significant bit of the mantissa if rounding control is set to round to $-\infty$ or round to 0 for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions; they are different for the FLDL2T instruction if round to $+\infty$ is specified. These changes were implemented for compatibility with IEEE Standard 754 for Floating-Point Arithmetic recommendations.

22.18.7.14 FSETPM Instruction

With the 32-bit x87 FPUs, the FSETPM instruction is treated as NOP (no operation). This instruction informs the Intel 287 math coprocessor that the processor is in protected mode. This change has no impact on existing software. The 32-bit x87 FPUs handle all addressing and exception-pointer information, whether in protected mode or not.

22.18.7.15 FXAM Instruction

With the 32-bit x87 FPUs, if the FPU encounters an empty register when executing the FXAM instruction, it not generate combinations of C0 through C3 equal to 1101 or 1111. The 16-bit IA-32 math coprocessors may generate these combinations, among others. This difference has no impact on existing software; it provides a performance upgrade to provide repeatable results.

22.18.7.16 FSAVE and FSTENV Instructions

With the 32-bit x87 FPUs, the address of a memory operand pointer stored by FSAVE or FSTENV is undefined if the previous floating-point instruction did not refer to memory

22.18.8 Transcendental Instructions

The floating-point results of the P6 family and Pentium processors for transcendental instructions in the core range may differ from the Intel486 processors by about 2 or 3 ulps (see "Transcendental Instruction Accuracy" in Chapter 8, "Programming with the x87 FPU," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Condition code flag C1 of the status word may differ as a result. The exact threshold for underflow and overflow will vary by a few ulps. The P6 family and Pentium processors' results will have a worst case error of less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The transcendental instructions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

Transcendental instructions may generate different results in the round-up flag (C1) on the 32-bit x87 FPUs. The round-up flag is undefined for these instructions on the 16-bit IA-32 math coprocessors. This difference has no impact on existing software.

22.18.9 Obsolete Instructions

The 8087 math coprocessor instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM are treated as integer NOP instructions in the 32-bit x87 FPUs. If these opcodes are detected in the instruction stream, no specific operation is performed and no internal states are affected.

22.18.10 WAIT/FWAIT Prefix Differences

On the Intel486 processor, when a WAIT/FWAIT instruction precedes a floating-point instruction (one which itself automatically synchronizes with the previous floating-point instruction), the WAIT/FWAIT instruction is treated as a no-op. Pending floating-point exceptions from a previous floating-point instruction are processed not on the WAIT/FWAIT instruction but on the floating-point instruction following the WAIT/FWAIT instruction. In such a case, the report of a floating-point exception may appear one instruction later on the Intel486 processor than on a P6 family or Pentium FPU, or on Intel 387 math coprocessor.

22.18.11 Operands Split Across Segments and/or Pages

On the P6 family, Pentium, and Intel486 processor FPUs, when the first half of an operand to be written is inside a page or segment and the second half is outside, a memory fault can cause the first half to be stored but not the second half. In this situation, the Intel 387 math coprocessor stores nothing.

22.18.12 FPU Instruction Synchronization

On the 32-bit x87 FPUs, all floating-point instructions are automatically synchronized; that is, the processor automatically waits until the previous floating-point instruction has completed before completing the next floating-point instruction. No explicit WAIT/FWAIT instructions are required to assure this synchronization. For the 8087 math coprocessors, explicit waits are required before each floating-point instruction to ensure synchronization. Although 8087 programs having explicit WAIT instructions execute perfectly on the 32-bit IA-32 processors without reassembly, these WAIT instructions are unnecessary.

22.19 SERIALIZING INSTRUCTIONS

Certain instructions have been defined to serialize instruction execution to ensure that modifications to flags, registers and memory are completed before the next instruction is executed (or in P6 family processor terminology "committed to machine state"). Because the P6 family processors use branch-prediction and out-of-order execution techniques to improve performance, instruction execution is not generally serialized until the results of an executed instruction are committed to machine state (see Chapter 2, "Intel® 64 and IA-32 Architectures," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

As a result, at places in a program or task where it is critical to have execution completed for all previous instructions before executing the next instruction (for example, at a branch, at the end of a procedure, or in multiprocessor dependent code), it is useful to add a serializing instruction. See Section 8.3, "Serializing Instructions," for more information on serializing instructions.

22.20 FPU AND MATH COPROCESSOR INITIALIZATION

Table 9-1 shows the states of the FPUs in the P6 family, Pentium, Intel486 processors and of the Intel 387 math coprocessor and Intel 287 coprocessor following a power-up, reset, or INIT, or following the execution of an FINIT/FNINIT instruction. The following is some additional compatibility information concerning the initialization of x87 FPUs and math coprocessors.

22.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization

Following an Intel386 processor reset, the processor identifies its coprocessor type (Intel® 287 or Intel® 387 DX math coprocessor) by sampling its ERROR# input some time after the falling edge of RESET# signal and before execution of the first floating-point instruction. The Intel 287 coprocessor keeps its ERROR# output in inactive state after hardware reset; the Intel 387 coprocessor keeps its ERROR# output in active state after hardware reset.

Upon hardware reset or execution of the FINIT/FNINIT instruction, the Intel 387 math coprocessor signals an error condition. The P6 family, Pentium, and Intel486 processors, like the Intel 287 coprocessor, do not.

22.20.2 Intel486 SX Processor and Intel 487 SX Math Coprocessor Initialization

When initializing an Intel486 SX processor and an Intel 487 SX math coprocessor, the initialization routine should check the presence of the math coprocessor and should set the FPU related flags (EM, MP, and NE) in control register CR0 accordingly (see Section 2.5, "Control Registers," for a complete description of these flags). Table 22-2 gives the recommended settings for these flags when the math coprocessor is present. The FSTCW instruction will give a value of FFFFH for the Intel486 SX microprocessor and 037FH for the Intel 487 SX math coprocessor.

Table 22-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System

CR0 Flags	Intel486 SX Processor Only	Intel 487 SX Math Coprocessor Present
EM	1	0
MP	0	1
NE	1	0, for MS-DOS* systems 1, for user-defined exception handler

The EM and MP flags in register CR0 are interpreted as shown in Table 22-3.

Table 22-3. EM and MP Flag Interpretation

EM	MP	Interpretation
0	0	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions ignore TS.
0	1	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions test TS.
1	0	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions ignore TS.
1	1	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions test TS.

Following is an example code sequence to initialize the system and check for the presence of Intel486 SX processor/Intel 487 SX math coprocessor.

```
fninit
fstcw mem_loc
mov ax, mem_loc
cmp ax, 037fh
jz Intel487_SX_Math_CoProcessor_present     ;ax=037fh
jmp Intel486_SX_microprocessor_present      ;ax=ffffh
```

If the Intel 487 SX math coprocessor is not present, the following code can be run to set the CR0 register for the Intel486 SX processor.

```
mov eax, cr0
and eax, fffffffdh  ;make MP=0
or eax, 0024h       ;make EM=1, NE=1
mov cr0, eax
```

This initialization will cause any floating-point instruction to generate a device not available exception (#NH), interrupt 7. The software emulation will then take control to execute these instructions. This code is not required if an Intel 487 SX math coprocessor is present in the system. In that case, the typical initialization routine for the Intel486 SX microprocessor will be adequate.

Also, when designing an Intel486 SX processor based system with an Intel 487 SX math coprocessor, timing loops should be independent of frequency and clocks per instruction. One way to attain this is to implement these loops in hardware and not in software (for example, BIOS).

22.21 CONTROL REGISTERS

The following sections identify the new control registers and control register flags and fields that were introduced to the 32-bit IA-32 in various processor families. See Figure 2-7 for the location of these flags and fields in the control registers.

The Pentium III processor introduced one new control flag in control register CR4:

- OSXMMEXCPT (bit 10) — The OS will set this bit if it supports unmasked SIMD floating-point exceptions.

The Pentium II processor introduced one new control flag in control register CR4:

- OSFXSR (bit 9) — The OS supports saving and restoring the Pentium III processor state during context switches.

The Pentium Pro processor introduced three new control flags in control register CR4:

- PAE (bit 5) — Physical address extension. Enables paging mechanism to reference extended physical addresses when set; restricts physical addresses to 32 bits when clear (see also: Section 22.22.1.1, "Physical Memory Addressing Extension").
- PGE (bit 7) — Page global enable. Inhibits flushing of frequently-used or shared pages on CR3 writes (see also: Section 22.22.1.2, "Global Pages").
- PCE (bit 8) — Performance-monitoring counter enable. Enables execution of the RDPMC instruction at any protection level.

The content of CR4 is 0H following a hardware reset.

Control register CR4 was introduced in the Pentium processor. This register contains flags that enable certain new extensions provided in the Pentium processor:

- VME — Virtual-8086 mode extensions. Enables support for a virtual interrupt flag in virtual-8086 mode (see Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode").
- PVI — Protected-mode virtual interrupts. Enables support for a virtual interrupt flag in protected mode (see Section 20.4, "Protected-Mode Virtual Interrupts").
- TSD — Time-stamp disable. Restricts the execution of the RDTSC instruction to procedures running at privileged level 0.
- DE — Debugging extensions. Causes an undefined opcode (#UD) exception to be generated when debug registers DR4 and DR5 are references for improved performance (see Section 22.23.3, "Debug Registers DR4 and DR5").
- PSE — Page size extensions. Enables 4-MByte pages with 32-bit paging when set (see Section 4.3, "32-Bit Paging").
- MCE — Machine-check enable. Enables the machine-check exception, allowing exception handling for certain hardware error conditions (see Chapter 15, "Machine-Check Architecture").

The Intel486 processor introduced five new flags in control register CR0:

- NE — Numeric error. Enables the normal mechanism for reporting floating-point numeric errors.
- WP — Write protect. Write-protects read-only pages against supervisor-mode accesses.
- AM — Alignment mask. Controls whether alignment checking is performed. Operates in conjunction with the AC (Alignment Check) flag.
- NW — Not write-through. Enables write-throughs and cache invalidation cycles when clear and disables invalidation cycles and write-throughs that hit in the cache when set.
- CD — Cache disable. Enables the internal cache when clear and disables the cache when set.

The Intel486 processor introduced two new flags in control register CR3:

- PCD — Page-level cache disable. The state of this flag is driven on the PCD# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PCD# pin is used to control caching in an external cache on a cycle-by-cycle basis.
- PWT — Page-level write-through. The state of this flag is driven on the PWT# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PWT# pin is used to control write through in an external cache on a cycle-by-cycle basis.

22.22 MEMORY MANAGEMENT FACILITIES

The following sections describe the new memory management facilities available in the various IA-32 processors and some compatibility differences.

22.22.1 New Memory Management Control Flags

The Pentium Pro processor introduced three new memory management features: physical memory addressing extension, the global bit in page-table entries, and general support for larger page sizes. These features are only available when operating in protected mode.

22.22.1.1 Physical Memory Addressing Extension

The new PAE (physical address extension) flag in control register CR4, bit 5, may enable additional address lines on the processor, allowing extended physical addresses. This option can only be used when paging is enabled, using a new page-table mechanism provided to support the larger physical address range (see Section 4.1, "Paging Modes and Control Bits").

22.22.1.2 Global Pages

The new PGE (page global enable) flag in control register CR4, bit 7, provides a mechanism for preventing frequently used pages from being flushed from the translation lookaside buffer (TLB). When this flag is set, frequently used pages (such as pages containing kernel procedures or common data tables) can be marked global by setting the global flag in a page-directory or page-table entry.

On a task switch or a write to control register CR3 (which normally causes the TLBs to be flushed), the entries in the TLB marked global are not flushed. Marking pages global in this manner prevents unnecessary reloading of the TLB due to TLB misses on frequently used pages. See Section 4.10, "Caching Translation Information" for a detailed description of this mechanism.

22.22.1.3 Larger Page Sizes

The P6 family processors support large page sizes. For 32-bit paging, this facility is enabled with the PSE (page size extension) flag in control register CR4, bit 4. When this flag is set, the processor supports either 4-KByte or 4-MByte page sizes. PAE paging and 4-level paging[1] support 2-MByte pages regardless of the value of CR4.PSE (see Section 4.4, "PAE Paging" and Section 4.5, "4-Level Paging"). See Chapter 4, "Paging," for more information about large page sizes.

22.22.2 CD and NW Cache Control Flags

The CD and NW flags in control register CR0 were introduced in the Intel486 processor. In the P6 family and Pentium processors, these flags are used to implement a writeback strategy for the data cache; in the Intel486 processor, they implement a write-through strategy. See Table 11-5 for a comparison of these bits on the P6 family, Pentium, and Intel486 processors. For complete information on caching, see Chapter 11, "Memory Cache Control."

22.22.3 Descriptor Types and Contents

Operating-system code that manages space in descriptor tables often contains an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors. Other values that were invalid on the Intel 286 processor may be valid on the 32-bit processors because uses for these bits have been defined.

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

22.22.4 Changes in Segment Descriptor Loads

On the Intel386 processor, loading a segment descriptor always causes a locked read and write to set the accessed bit of the descriptor. On the P6 family, Pentium, and Intel486 processors, the locked read and write occur only if the bit is not already set.

22.23 DEBUG FACILITIES

The P6 family and Pentium processors include extensions to the Intel486 processor debugging support for breakpoints. To use the new breakpoint features, it is necessary to set the DE flag in control register CR4.

22.23.1 Differences in Debug Register DR6

It is not possible to write a 1 to reserved bit 12 in debug status register DR6 on the P6 family and Pentium processors; however, it is possible to write a 1 in this bit on the Intel486 processor. See Table 9-1 for the different setting of this register following a power-up or hardware reset.

22.23.2 Differences in Debug Register DR7

The P6 family and Pentium processors determines the type of breakpoint access by the R/W0 through R/W3 fields in debug control register DR7 as follows:

00 Break on instruction execution only.

01 Break on data writes only.

10 Undefined if the DE flag in control register CR4 is cleared; break on I/O reads or writes but not instruction fetches if the DE flag in control register CR4 is set.

11 Break on data reads or writes but not instruction fetches.

On the P6 family and Pentium processors, reserved bits 11, 12, 14 and 15 are hard-wired to 0. On the Intel486 processor, however, bit 12 can be set. See Table 9-1 for the different settings of this register following a power-up or hardware reset.

22.23.3 Debug Registers DR4 and DR5

Although the DR4 and DR5 registers are documented as reserved, previous generations of processors aliased references to these registers to debug registers DR6 and DR7, respectively. When debug extensions are not enabled (the DE flag in control register CR4 is cleared), the P6 family and Pentium processors remain compatible with existing software by allowing these aliased references. When debug extensions are enabled (the DE flag is set), attempts to reference registers DR4 or DR5 will result in an invalid-opcode exception (#UD).

22.24 RECOGNITION OF BREAKPOINTS

For the Pentium processor, it is recommended that debuggers execute the LGDT instruction before returning to the program being debugged to ensure that breakpoints are detected. This operation does not need to be performed on the P6 family, Intel486, or Intel386 processors.

The implementation of test registers on the Intel486 processor used for testing the cache and TLB has been redesigned using MSRs on the P6 family and Pentium processors. (Note that MSRs used for this function are different on the P6 family and Pentium processors.) The MOV to and from test register instructions generate invalid-opcode exceptions (#UD) on the P6 family processors.

22.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS

This section describes the new exceptions and exception conditions added to the 32-bit IA-32 processors and implementation differences in existing exception handling. See Chapter 6, "Interrupt and Exception Handling," for a detailed description of the IA-32 exceptions.

The Pentium III processor introduced new state with the XMM registers. Computations involving data in these registers can produce exceptions. A new MXCSR control/status register is used to determine which exception or exceptions have occurred. When an exception associated with the XMM registers occurs, an interrupt is generated.

- SIMD floating-point exception (#XM, interrupt 19) — New exceptions associated with the SIMD floating-point registers and resulting computations.

No new exceptions were added with the Pentium Pro and Pentium II processors. The set of available exceptions is the same as for the Pentium processor. However, the following exception condition was added to the IA-32 with the Pentium Pro processor:

- Machine-check exception (#MC, interrupt 18) — New exception conditions. Many exception conditions have been added to the machine-check exception and a new architecture has been added for handling and reporting on hardware errors. See Chapter 15, "Machine-Check Architecture," for a detailed description of the new conditions.

The following exceptions and/or exception conditions were added to the IA-32 with the Pentium processor:

- Machine-check exception (#MC, interrupt 18) — New exception. This exception reports parity and other hardware errors. It is a model-specific exception and may not be implemented or implemented differently in future processors. The MCE flag in control register CR4 enables the machine-check exception. When this bit is clear (which it is at reset), the processor inhibits generation of the machine-check exception.
- General-protection exception (#GP, interrupt 13) — New exception condition added. An attempt to write a 1 to a reserved bit position of a special register causes a general-protection exception to be generated.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. When a 1 is detected in any of the reserved bit positions of a page-table entry, page-directory entry, or page-directory pointer during address translation, a page-fault exception is generated.

The following exception was added to the Intel486 processor:

- Alignment-check exception (#AC, interrupt 17) — New exception. Reports unaligned memory references when alignment checking is being performed.

The following exceptions and/or exception conditions were added to the Intel386 processor:

- Divide-error exception (#DE, interrupt 0)
  - Change in exception handling. Divide-error exceptions on the Intel386 processors always leave the saved CS:IP value pointing to the instruction that failed. On the 8086 processor, the CS:IP value points to the next instruction.
  - Change in exception handling. The Intel386 processors can generate the largest negative number as a quotient for the IDIV instruction (80H and 8000H). The 8086 processor generates a divide-error exception instead.
- Invalid-opcode exception (#UD, interrupt 6) — New exception condition added. Improper use of the LOCK instruction prefix can generate an invalid-opcode exception.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. If paging is enabled in a 16-bit program, a page-fault exception can be generated as follows. Paging can be used in a system with 16-bit tasks if all tasks use the same page directory. Because there is no place in a 16-bit TSS to store the PDBR register, switching to a 16-bit task does not change the value of the PDBR register. Tasks ported from the Intel 286 processor should be given 32-bit TSSs so they can make full use of paging.
- General-protection exception (#GP, interrupt 13) — New exception condition added. The Intel386 processor sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. A general-protection exception is generated if the limit on instruction length is violated. The 8086 processor has no instruction length limit.

22.25.1 Machine-Check Architecture

The Pentium Pro processor introduced a new architecture to the IA-32 for handling and reporting on machine-check exceptions. This machine-check architecture (described in detail in Chapter 15, "Machine-Check Architecture") greatly expands the ability of the processor to report on internal hardware errors.

22.25.2 Priority of Exceptions

The priority of exceptions are broken down into several major categories:

1. Traps on the previous instruction
2. External interrupts
3. Faults on fetching the next instruction
4. Faults in decoding the next instruction
5. Faults on executing an instruction

There are no changes in the priority of these major categories between the different processors, however, exceptions within these categories are implementation dependent and may change from processor to processor.

22.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers

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

Table 22-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
#XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPD2PI, CVTTPD2PI				

Table 22-5. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPI2PS, CVTPS2PI, CVTTPS2PI				

Table 22-6. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
#MF[1]	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	CVTPI2PD				

NOTES:

1. Applies to “CVTPI2PD xmm, mm” but not “CVTPI2PD xmm, m64”.

Table 22-7. Exception Conditions for SIMD/MMX Instructions with Memory Reference

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	PABSB, PABSD, PABSW, PACKSSWB, PACKSSDW, PACKUSWB, PADDB, PADDD, PADDQ, PADDW, PADDSB, PADDSW, PADDUSB, PADDUSW, PALIGNR, PAND, PANDN, PAVGB, PAVGW, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PHADDD, PHADDW, PHADDSW, PHSUBD, PHSUBW, PHSUBSW, PINSRW, PMADDUBSW, PMADDWD, PMAXSW, PMAXUB, PMINSW, PMINUB, PMULHRSW, PMULHUW, PMULHW, PMULLW, PMULUDQ, PSADBW, PSHUFB, PSHUFW, PSIGNB PSIGND PSIGNW, PSLLW, PSLLD, PSLLQ, PSRAD, PSRAW, PSRLW, PSRLD, PSRLQ, PSUBB, PSUBD, PSUBQ, PSUBW, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PXOR					

Table 22-8. Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1. If ModR/M.mod $\neq$ 11b[1]
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
#GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the destination operand is in a non-writable segment.[2] If the DS, ES, FS, or GS register contains a NULL segment selector.[3]
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
#AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	MASKMOVQ, MOVNTQ, “MOVQ (mmreg)”				

NOTES:

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

Table 22-9. Exception Conditions for Legacy SIMD/MMX Instructions without Memory Reference

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM			X	X	If CR0.TS[bit 3]=1
Applicable Instructions	PEXTRW, PMOVMSKB				

22.26 INTERRUPTS

The following differences in handling interrupts are found among the IA-32 processors.

22.26.1 Interrupt Propagation Delay

External hardware interrupts may be recognized on different instruction boundaries on the P6 family, Pentium, Intel486, and Intel386 processors, due to the superscaler designs of the P6 family and Pentium processors. Therefore, the EIP pushed onto the stack when servicing an interrupt may be different for the P6 family, Pentium, Intel486, and Intel386 processors.

22.26.2 NMI Interrupts

After an NMI interrupt is recognized by the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors, the NMI interrupt is masked until the first IRET instruction is executed, unlike the 8086 processor.

22.26.3 IDT Limit

The LIDT instruction can be used to set a limit on the size of the IDT. A double-fault exception (#DF) is generated if an interrupt or exception attempts to read a vector beyond the limit. Shutdown then occurs on the 32-bit IA-32 processors if the double-fault handler vector is beyond the limit. (The 8086 processor does not have a shutdown mode nor a limit.)

22.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in this book as the **local APIC**, was introduced into the IA-32 processors with the Pentium processor (beginning with the 735/90 and 815/100 models) and is included in the Pentium 4, Intel Xeon, and P6 family processors. The features and functions of the local APIC are derived from the Intel 82489DX external APIC, which was used with the Intel486 and early Pentium processors. Additional refinements of the local APIC architecture were incorporated in the Pentium 4 and Intel Xeon processors.

22.27.1 Software Visible Differences Between the Local APIC and the 82489DX

The following features in the local APIC features differ from those found in the 82489DX external APIC:

- When the local APIC is disabled by clearing the APIC software enable/disable flag in the spurious-interrupt vector MSR, the state of its internal registers are unaffected, except that the mask bits in the LVT are all set to block local interrupts to the processor. Also, the local APIC ceases accepting IPIs except for INIT, SMI, NMI, and start-up IPIs. In the 82489DX, when the local unit is disabled, all the internal registers including the IRR, ISR and TMR are cleared and the mask bits in the LVT are set. In this state, the 82489DX local unit will accept only the reset deassert message.
- In the local APIC, NMI and INIT (except for INIT deassert) are always treated as edge triggered interrupts, even if programmed otherwise. In the 82489DX, these interrupts are always level triggered.
- In the local APIC, IPIs generated through the ICR are always treated as edge triggered (except INIT Deassert). In the 82489DX, the ICR can be used to generate either edge or level triggered IPIs.
- In the local APIC, the logical destination register supports 8 bits; in the 82489DX, it supports 32 bits.
- In the local APIC, the APIC ID register is 4 bits wide; in the 82489DX, it is 8 bits wide.
- The remote read delivery mode provided in the 82489DX and local APIC for Pentium processors is not supported in the local APIC in the Pentium 4, Intel Xeon, and P6 family processors.
- For the 82489DX, in the lowest priority delivery mode, all the target local APICs specified by the destination field participate in the lowest priority arbitration. For the local APIC, only those local APICs which have free interrupt slots will participate in the lowest priority arbitration.

22.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors

The local APIC in the Pentium and P6 family processors have the following new features not found in the 82489DX external APIC.

- Cluster addressing is supported in logical destination mode.
- Focus processor checking can be enabled/disabled.
- Interrupt input signal polarity can be programmed for the LINT0 and LINT1 pins.
- An SMI IPI is supported through the ICR and I/O redirection table.
- An error status register is incorporated into the LVT to log and report APIC errors.

In the P6 family processors, the local APIC incorporates an additional LVT register to handle performance monitoring counter interrupts.

22.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors

The local APIC in the Pentium 4 and Intel Xeon processors has the following new features not found in the P6 family and Pentium processors and in the 82489DX.

- The local APIC ID is extended to 8 bits.
- An thermal sensor register is incorporated into the LVT to handle thermal sensor interrupts.
- The the ability to deliver lowest-priority interrupts to a focus processor is no longer supported.
- The flat cluster logical destination mode is not supported.

22.28 TASK SWITCHING AND TSS

This section identifies the implementation differences of task switching, additions to the TSS and the handling of TSSs and TSS segment selectors.

22.28.1 P6 Family and Pentium Processor TSS

When the virtual mode extensions are enabled (by setting the VME flag in control register CR4), the TSS in the P6 family and Pentium processors contain an interrupt redirection bit map, which is used in virtual-8086 mode to redirect interrupts back to an 8086 program.

22.28.2 TSS Selector Writes

During task state saves, the Intel486 processor writes 2-byte segment selectors into a 32-bit TSS, leaving the upper 16 bits undefined. For performance reasons, the P6 family and Pentium processors write 4-byte segment selectors into the TSS, with the upper 2 bytes being 0. For compatibility reasons, code should not depend on the value of the upper 16 bits of the selector in the TSS.

22.28.3 Order of Reads/Writes to the TSS

The order of reads and writes into the TSS is processor dependent. The P6 family and Pentium processors may generate different page-fault addresses in control register CR2 in the same TSS area than the Intel486 and Intel386 processors, if a TSS crosses a page boundary (which is not recommended).

22.28.4 Using A 16-Bit TSS with 32-Bit Constructs

Task switches using 16-bit TSSs should be used only for pure 16-bit code. Any new code written using 32-bit constructs (operands, addressing, or the upper word of the EFLAGS register) should use only 32-bit TSSs. This is due to the fact that the 32-bit processors do not save the upper 16 bits of EFLAGS to a 16-bit TSS. A task switch back to a 16-bit task that was executing in virtual mode will never re-enable the virtual mode, as this flag was not saved in the upper half of the EFLAGS value in the TSS. Therefore, it is strongly recommended that any code using 32-bit constructs use a 32-bit TSS to ensure correct behavior in a multitasking environment.

22.28.5 Differences in I/O Map Base Addresses

The Intel486 processor considers the TSS segment to be a 16-bit segment and wraps around the 64K boundary. Any I/O accesses check for permission to access this I/O address at the I/O base address plus the I/O offset. If the I/O map base address exceeds the specified limit of 0DFFFH, an I/O access will wrap around and obtain the permission for the I/O address at an incorrect location within the TSS. A TSS limit violation does not occur in this situation on the Intel486 processor. However, the P6 family and Pentium processors consider the TSS to be a 32-bit segment and a limit violation occurs when the I/O base address plus the I/O offset is greater than the TSS limit. By following the recommended specification for the I/O base address to be less than 0DFFFH, the Intel486 processor will not wrap around and access incorrect locations within the TSS for I/O port validation and the P6 family and Pentium processors will not experience general-protection exceptions (#GP). Figure 22-1 demonstrates the different areas accessed by the Intel486 and the P6 family and Pentium processors.

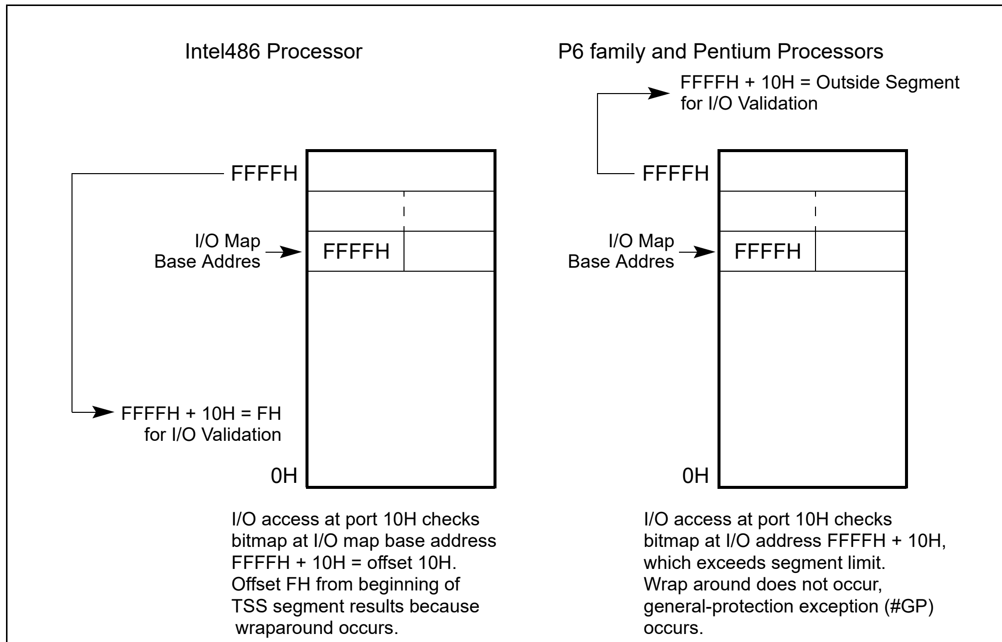


Figure 22-1. I/O Map Base Address Differences

22.29 CACHE MANAGEMENT

The P6 family processors include two levels of internal caches: L1 (level 1) and L2 (level 2). The L1 cache is divided into an instruction cache and a data cache; the L2 cache is a general-purpose cache. See Section 11.1, "Internal Caches, TLBs, and Buffers," for a description of these caches. (Note that although the Pentium II processor L2 cache is physically located on a separate chip in the cassette, it is considered an internal cache.)

The Pentium processor includes separate level 1 instruction and data caches. The data cache supports a writeback (or alternatively write-through, on a line by line basis) policy for memory updates.

The Intel486 processor includes a single level 1 cache for both instructions and data.

The meaning of the CD and NW flags in control register CR0 have been redefined for the P6 family and Pentium processors. For these processors, the recommended value (00B) enables writeback for the data cache of the Pentium processor and for the L1 data cache and L2 cache of the P6 family processors. In the Intel486 processor, setting these flags to (00B) enables write-through for the cache.

External system hardware can force the Pentium processor to disable caching or to use the write-through cache policy should that be required. In the P6 family processors, the MTRRs can be used to override the CD and NW flags (see Table 11-6).

The P6 family and Pentium processors support page-level cache management in the same manner as the Intel486 processor by using the PCD and PWT flags in control register CR3, the page-directory entries, and the page-table entries. The Intel486 processor, however, is not affected by the state of the PWT flag since the internal cache of the Intel486 processor is a write-through cache.

22.29.1 Self-Modifying Code with Cache Enabled

On the Intel486 processor, a write to an instruction in the cache will modify it in both the cache and memory. If the instruction was prefetched before the write, however, the old version of the instruction could be the one executed. To prevent this problem, it is necessary to flush the instruction prefetch unit of the Intel486 processor by coding a jump instruction immediately after any write that modifies an instruction. The P6 family and Pentium processors, however, check whether a write may modify an instruction that has been prefetched for execution. This check is based on the linear address of the instruction. If the linear address of an instruction is found to be present in the

prefetch queue, the P6 family and Pentium processors flush the prefetch queue, eliminating the need to code a jump instruction after any writes that modify an instruction.

Because the linear address of the write is checked against the linear address of the instructions that have been prefetched, special care must be taken for self-modifying code to work correctly when the physical addresses of the instruction and the written data are the same, but the linear addresses differ. In such cases, it is necessary to execute a serializing operation to flush the prefetch queue after the write and before executing the modified instruction. See Section 8.3, "Serializing Instructions," for more information on serializing instructions.

NOTE

The check on linear addresses described above is not in practice a concern for compatibility. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. System software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction must execute a serializing operation, such as IRET, before the modified instruction is executed.

22.29.2 Disabling the L3 Cache

A unified third-level (L3) cache in processors based on Intel NetBurst microarchitecture (see Section 11.1, "Internal Caches, TLBs, and Buffers") provides the third-level cache disable flag, bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches (see Section 11.5.4, "Disabling and Enabling the L3 Cache"). The third-level cache disable flag applies only to processors based on Intel NetBurst microarchitecture. Processors with L3 and based on other microarchitectures do not support the third-level cache disable flag.

22.30 PAGING

This section identifies enhancements made to the paging mechanism and implementation differences in the paging mechanism for various IA-32 processors.

22.30.1 Large Pages

The Pentium processor extended the memory management/paging facilities of the IA-32 to allow large (4 MBytes) pages sizes (see Section 4.3, "32-Bit Paging"). The first P6 family processor (the Pentium Pro processor) added a 2 MByte page size to the IA-32 in conjunction with the physical address extension (PAE) feature (see Section 4.4, "PAE Paging").

The availability of large pages with 32-bit paging on any IA-32 processor can be determined via feature bit 3 (PSE) of register EDX after the CPUID instruction has been execution with an argument of 1. (Large pages are always available with PAE paging and 4-level paging.) Intel processors that do not support the CPUID instruction support only 32-bit paging and do not support page size enhancements. (See "CPUID—CPU Identification" in Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for more information on the CPUID instruction.)

22.30.2 PCD and PWT Flags

The PCD and PWT flags were introduced to the IA-32 in the Intel486 processor to control the caching of pages:

- PCD (page-level cache disable) flag—Controls caching on a page-by-page basis.
- PWT (page-level write-through) flag—Controls the write-through/writeback caching policy on a page-by-page basis. Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag.

22.30.3 Enabling and Disabling Paging

Paging is enabled and disabled by loading a value into control register CR0 that modifies the PG flag. For backward and forward compatibility with all IA-32 processors, Intel recommends that the following operations be performed when enabling or disabling paging:

1. Execute a MOV CR0, REG instruction to either set (enable paging) or clear (disable paging) the PG flag.
2. Execute a near JMP instruction.

The sequence bounded by the MOV and JMP instructions should be identity mapped (that is, the instructions should reside on a page whose linear and physical addresses are identical).

For the P6 family processors, the MOV CR0, REG instruction is serializing, so the jump operation is not required. However, for backwards compatibility, the JMP instruction should still be included.

22.31 STACK OPERATIONS AND SUPERVISOR SOFTWARE

This section identifies the differences in the stack mechanism for the various IA-32 processors.

22.31.1 Selector Pushes and Pops

When pushing a segment selector onto the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors decrement the ESP register by the operand size and then write 2 bytes. If the operand size is 32-bits, the upper two bytes of the write are not modified. The Pentium processor decrements the ESP register by the operand size and determines the size of the write by the operand size. If the operand size is 32-bits, the upper two bytes are written as 0s.

When popping a segment selector from the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors read 2 bytes and increment the ESP register by the operand size of the instruction. The Pentium processor determines the size of the read from the operand size and increments the ESP register by the operand size.

It is possible to align a 32-bit selector push or pop such that the operation generates an exception on a Pentium processor and not on an Pentium 4, Intel Xeon, P6 family, or Intel486 processor. This could occur if the third and/or fourth byte of the operation lies beyond the limit of the segment or if the third and/or fourth byte of the operation is locate on a non-present or inaccessible page.

For a POP-to-memory instruction that meets the following conditions:

- The stack segment size is 16-bit.
- Any 32-bit addressing form with the SIB byte specifying ESP as the base register.
- The initial stack pointer is FFFCH (32-bit operand) or FFFEH (16-bit operand) and will wrap around to 0H as a result of the POP operation.

The result of the memory write is implementation-specific. For example, in P6 family processors, the result of the memory write is SS:0H plus any scaled index and displacement. In Pentium processors, the result of the memory write may be either a stack fault (real mode or protected mode with stack segment size of 64 KByte), or write to SS:10000H plus any scaled index and displacement (protected mode and stack segment size exceeds 64 KByte).

22.31.2 Error Code Pushes

The Intel486 processor implements the error code pushed on the stack as a 16-bit value. When pushed onto a 32-bit stack, the Intel486 processor only pushes 2 bytes and updates ESP by 4. The P6 family and Pentium processors' error code is a full 32 bits with the upper 16 bits set to zero. The P6 family and Pentium processors, therefore, push 4 bytes and update ESP by 4. Any code that relies on the state of the upper 16 bits may produce inconsistent results.

22.31.3 Fault Handling Effects on the Stack

During the handling of certain instructions, such as CALL and PUSHA, faults may occur in different sequences for the different processors. For example, during far calls, the Intel486 processor pushes the old CS and EIP before a possible branch fault is resolved. A branch fault is a fault from a branch instruction occurring from a segment limit or access rights violation. If a branch fault is taken, the Intel486 and P6 family processors will have corrupted memory below the stack pointer. However, the ESP register is backed up to make the instruction restartable. The P6 family processors issue the branch before the pushes. Therefore, if a branch fault does occur, these processors do not corrupt memory below the stack pointer. This implementation difference, however, does not constitute a compatibility problem, as only values at or above the stack pointer are considered to be valid. Other operations that encounter faults may also corrupt memory below the stack pointer and this behavior may vary on different implementations.

22.31.4 Interlevel RET/IRET From a 16-Bit Interrupt or Call Gate

If a call or interrupt is made from a 32-bit stack environment through a 16-bit gate, only 16 bits of the old ESP can be pushed onto the stack. On the subsequent RET/IRET, the 16-bit ESP is popped but the full 32-bit ESP is updated since control is being resumed in a 32-bit stack environment. The Intel486 processor writes the SS selector into the upper 16 bits of ESP. The P6 family and Pentium processors write zeros into the upper 16 bits.

22.32 MIXING 16- AND 32-BIT SEGMENTS

The features of the 16-bit Intel 286 processor are an object-code compatible subset of those of the 32-bit IA-32 processors. The D (default operation size) flag in segment descriptors indicates whether the processor treats a code or data segment as a 16-bit or 32-bit segment; the B (default stack size) flag in segment descriptors indicates whether the processor treats a stack segment as a 16-bit or 32-bit segment.

The segment descriptors used by the Intel 286 processor are supported by the 32-bit IA-32 processors if the Intel-reserved word (highest word) of the descriptor is clear. On the 32-bit IA-32 processors, this word includes the upper bits of the base address and the segment limit.

The segment descriptors for data segments, code segments, local descriptor tables (there are no descriptors for global descriptor tables), and task gates are the same for the 16- and 32-bit processors. Other 16-bit descriptors (TSS segment, call gate, interrupt gate, and trap gate) are supported by the 32-bit processors.

The 32-bit processors also have descriptors for TSS segments, call gates, interrupt gates, and trap gates that support the 32-bit architecture. Both kinds of descriptors can be used in the same system.

For those segment descriptors common to both 16- and 32-bit processors, clear bits in the reserved word cause the 32-bit processors to interpret these descriptors exactly as an Intel 286 processor does, that is:

- Base Address — The upper 8 bits of the 32-bit base address are clear, which limits base addresses to 24 bits.
- Limit — The upper 4 bits of the limit field are clear, restricting the value of the limit field to 64 KBytes.
- Granularity bit — The G (granularity) flag is clear, indicating the value of the 16-bit limit is interpreted in units of 1 byte.
- Big bit — In a data-segment descriptor, the B flag is clear in the segment descriptor used by the 32-bit processors, indicating the segment is no larger than 64 KBytes.
- Default bit — In a code-segment descriptor, the D flag is clear, indicating 16-bit addressing and operands are the default. In a stack-segment descriptor, the D flag is clear, indicating use of the SP register (instead of the ESP register) and a 64-KByte maximum segment limit.

For information on mixing 16- and 32-bit code in applications, see Chapter 21, “Mixing 16-Bit and 32-Bit Code.”

22.33 SEGMENT AND ADDRESS WRAPAROUND

This section discusses differences in segment and address wraparound between the P6 family, Pentium, Intel486, Intel386, Intel 286, and 8086 processors.

22.33.1 Segment Wraparound

On the 8086 processor, an attempt to access a memory operand that crosses offset 65,535 or 0FFFFH or offset 0 (for example, moving a word to offset 65,535 or pushing a word when the stack pointer is set to 1) causes the offset to wrap around modulo 65,536 or 010000H. With the Intel 286 processor, any base and offset combination that addresses beyond 16 MBytes wraps around to the 1 MByte of the address space. The P6 family, Pentium, Intel486, and Intel386 processors in real-address mode generate an exception in these cases:

- A general-protection exception (#GP) if the segment is a data segment (that is, if the CS, DS, ES, FS, or GS register is being used to address the segment).
- A stack-fault exception (#SS) if the segment is a stack segment (that is, if the SS register is being used).

An exception to this behavior occurs when a stack access is data aligned, and the stack pointer is pointing to the last aligned piece of data that size at the top of the stack (ESP is FFFFFFFCH). When this data is popped, no segment limit violation occurs and the stack pointer will wrap around to 0.

The address space of the P6 family, Pentium, and Intel486 processors may wraparound at 1 MByte in real-address mode. An external A20M# pin forces wraparound if enabled. On Intel 8086 processors, it is possible to specify addresses greater than 1 MByte. For example, with a selector value FFFFH and an offset of FFFFH, the effective address would be 10FFEFH (1 MByte plus 65519 bytes). The 8086 processor, which can form addresses up to 20 bits long, truncates the uppermost bit, which "wraps" this address to FFEFH. However, the P6 family, Pentium, and Intel486 processors do not truncate this bit if A20M# is not enabled.

If a stack operation wraps around the address limit, shutdown occurs. (The 8086 processor does not have a shut-down mode or a limit.)

The behavior when executing near the limit of a 4-GByte selector (limit = FFFFFFFFH) is different between the Pentium Pro and the Pentium 4 family of processors. On the Pentium Pro, instructions which cross the limit -- for example, a two byte instruction such as INC EAX that is encoded as FFH C0H starting exactly at the limit faults for a segment violation (a one byte instruction at FFFFFFFFH does not cause an exception). Using the Pentium 4 micro-processor family, neither of these situations causes a fault.

Segment wraparound and the functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

22.34 STORE BUFFERS AND MEMORY ORDERING

The Pentium 4, Intel Xeon, and P6 family processors provide a store buffer for temporary storage of writes (stores) to memory (see Section 11.10, "Store Buffer"). Writes stored in the store buffer(s) are always written to memory in program order, with the exception of "fast string" store operations (see Section 8.2.4, "Fast-String Operation and Out-of-Order Stores").

The Pentium processor has two store buffers, one corresponding to each of the pipelines. Writes in these buffers are always written to memory in the order they were generated by the processor core.

It should be noted that only memory writes are buffered and I/O writes are not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors do not synchronize the completion of memory writes on the bus and instruction execution after a write. An I/O, locked, or serializing instruction needs to be executed to synchronize writes with the next instruction (see Section 8.3, "Serializing Instructions").

The Pentium 4, Intel Xeon, and P6 family processors use processor ordering to maintain consistency in the order that data is read (loaded) and written (stored) in a program and the order the processor actually carries out the reads and writes. With this type of ordering, reads can be carried out speculatively and in any order, reads can pass buffered writes, and writes to memory are always carried out in program order. (See Section 8.2, "Memory Ordering," for more information about processor ordering.) The Pentium III processor introduced a new instruction to serialize writes and make them globally visible. Memory ordering issues can arise between a producer and a consumer of data. The SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

No re-ordering of reads occurs on the Pentium processor, except under the condition noted in Section 8.2.1, "Memory Ordering in the Intel® Pentium® and Intel486™ Processors," and in the following paragraph describing the Intel486 processor.

Specifically, the store buffers are flushed before the IN instruction is executed. No reads (as a result of cache miss) are reordered around previously generated writes sitting in the store buffers. The implication of this is that the store buffers will be flushed or emptied before a subsequent bus cycle is run on the external bus.

On both the Intel486 and Pentium processors, under certain conditions, a memory read will go onto the external bus before the pending memory writes in the buffer even though the writes occurred earlier in the program execution. A memory read will only be reordered in front of all writes pending in the buffers if all writes pending in the buffers are cache hits and the read is a cache miss. Under these conditions, the Intel486 and Pentium processors will not read from an external memory location that needs to be updated by one of the pending writes.

During a locked bus cycle, the Intel486 processor will always access external memory, it will never look for the location in the on-chip cache. All data pending in the Intel486 processor's store buffers will be written to memory before a locked cycle is allowed to proceed to the external bus. Thus, the locked bus cycle can be used for eliminating the possibility of reordering read cycles on the Intel486 processor. The Pentium processor does check its cache on a read-modify-write access and, if the cache line has been modified, writes the contents back to memory before locking the bus. The P6 family processors write to their cache on a read-modify-write operation (if the access does not split across a cache line) and does not write back to system memory. If the access does split across a cache line, it locks the bus and accesses system memory.

I/O reads are never reordered in front of buffered memory writes on an IA-32 processor. This ensures an update of all memory locations before reading the status from an I/O device.

22.35 BUS LOCKING

The Intel 286 processor performs the bus locking differently than the Intel P6 family, Pentium, Intel486, and Intel386 processors. Programs that use forms of memory locking specific to the Intel 286 processor may not run properly when run on later processors.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and Intel 286 configurations lock the entire physical memory space. Programmers should not depend on this.

On the Intel 286 processor, the LOCK prefix is sensitive to IOPL. If the CPL is greater than the IOPL, a general-protection exception (#GP) is generated. On the Intel386 DX, Intel486, and Pentium, and P6 family processors, no check against IOPL is performed.

The Pentium processor automatically asserts the LOCK# signal when acknowledging external interrupts. After signaling an interrupt request, an external interrupt controller may use the data bus to send the interrupt vector to the processor. After receiving the interrupt request signal, the processor asserts LOCK# to insure that no other data appears on the data bus until the interrupt vector is received. This bus locking does not occur on the P6 family processors.

22.36 BUS HOLD

Unlike the 8086 and Intel 286 processors, but like the Intel386 and Intel486 processors, the P6 family and Pentium processors respond to requests for control of the bus from other potential bus masters, such as DMA controllers, between transfers of parts of an unaligned operand, such as two words which form a doubleword. Unlike the Intel386 processor, the P6 family, Pentium and Intel486 processors respond to bus hold during reset initialization.

22.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32

Certain extensions to the IA-32 are specific to a processor or family of IA-32 processors and may not be implemented or implemented in the same way in future processors. The following sections describe these model-specific extensions. The CPUID instruction indicates the availability of some of the model-specific features.

22.37.1 Model-Specific Registers

The Pentium processor introduced a set of model-specific registers (MSRs) for use in controlling hardware functions and performance monitoring. To access these MSRs, two new instructions were added to the IA-32 architecture: read MSR (RDMSR) and write MSR (WRMSR). The MSRs in the Pentium processor are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

The P6 family processors greatly increased the number of MSRs available to software. See Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for a complete list of the available MSRs. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine-check architecture, and the MTRRs. These registers are accessible using the RDMSR and WRMSR instructions. Specific information on some of these new MSRs is provided in the following sections. As with the Pentium processor MSR, the P6 family processor MSRs are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

22.37.2 RDMSR and WRMSR Instructions

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions recognize a much larger number of model-specific registers in the P6 family processors. (See "RDMSR—Read from Model Specific Register" and "WRMSR—Write to Model Specific Register" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D* for more information.)

22.37.3 Memory Type Range Registers

Memory type range registers (MTRRs) are a new feature introduced into the IA-32 in the Pentium Pro processor. MTRRs allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped I/O.

MTRRs are MSRs that contain an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal memory map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), uncacheable, speculatable, write-combining (WC), write-through (WT), write-protected (WP), and writeback (WB).

Earlier IA-32 processors (such as the Intel486 and Pentium processors) used the KEN# (cache enable) pin and external logic to maintain an external memory map and signal cacheable accesses to the processor. The MTRR mechanism simplifies hardware designs by eliminating the KEN# pin and the external logic required to drive it.

See Chapter 9, "Processor Management and Initialization," and Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for more information on the MTRRs.

22.37.4 Machine-Check Exception and Architecture

The Pentium processor introduced a new exception called the machine-check exception (#MC, interrupt 18). This exception is used to detect hardware-related errors, such as a parity error on a read cycle.

The P6 family processors extend the types of errors that can be detected and that generate a machine-check exception. It also provides a new machine-check architecture for recording information about a machine-check error and provides extended recovery capability.

The machine-check architecture provides several banks of reporting registers for recording machine-check errors. Each bank of registers is associated with a specific hardware unit in the processor. The primary focus of the machine checks is on bus and interconnect operations; however, checks are also made of translation lookaside buffer (TLB) and cache operations.

The machine-check architecture can correct some errors automatically and allow for reliable restart of instruction execution. It also collects sufficient information for software to use in correcting other machine errors not corrected by hardware.

See Chapter 15, "Machine-Check Architecture," for more information on the machine-check exception and the machine-check architecture.

22.37.5 Performance-Monitoring Counters

The P6 family and Pentium processors provide two performance-monitoring counters for use in monitoring internal hardware operations. The number of performance monitoring counters and associated programming interfaces may be implementation specific for Pentium 4 processors, Pentium M processors. Later processors may have implemented these as part of an architectural performance monitoring feature. The architectural and non-architectural performance monitoring interfaces for different processor families are described in Chapter 18, "Performance Monitoring,". Chapter 19, "Performance Monitoring Events." lists all the events that can be counted for architectural performance monitoring events and non-architectural events. The counters are set up, started, and stopped using two MSRs and the RDMSR and WRMSR instructions. For the P6 family processors, the current count for a particular counter can be read using the new RDPMC instruction.

The performance-monitoring counters are useful for debugging programs, optimizing code, diagnosing system failures, or refining hardware designs. See Chapter 18, "Performance Monitoring," for more information on these counters.

22.38 TWO WAYS TO RUN INTEL 286 PROCESSOR TASKS

When porting 16-bit programs to run on 32-bit IA-32 processors, there are two approaches to consider:

- Porting an entire 16-bit software system to a 32-bit processor, complete with the old operating system, loader, and system builder. Here, all tasks will have 16-bit TSSs. The 32-bit processor is being used as if it were a faster version of the 16-bit processor.
- Porting selected 16-bit applications to run in a 32-bit processor environment with a 32-bit operating system, loader, and system builder. Here, the TSSs used to represent 286 tasks should be changed to 32-bit TSSs. It is possible to mix 16 and 32-bit TSSs, but the benefits are small and the problems are great. All tasks in a 32-bit software system should have 32-bit TSSs. It is not necessary to change the 16-bit object modules themselves; TSSs are usually constructed by the operating system, by the loader, or by the system builder. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," for more detailed information about mixing 16-bit and 32-bit code.

Because the 32-bit processors use the contents of the reserved word of 16-bit segment descriptors, 16-bit programs that place values in this word may not run correctly on the 32-bit processors.

22.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS

Table 22-10 shows the state of the flags and other registers following power-up for the Pentium, Pentium Pro and Pentium 4 processors. The state of control register CR0 is 60000010H (see Figure 9-1 "Contents of CR0 Register after Reset" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). This places the processor in real-address mode with paging disabled.

Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H

Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	00000FxxH	000n06xxH[3]	000005xxH
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5]	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged
x87 FPU Control Word[5]	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH
x87 FPU Status Word[5]	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Tag Word[5]	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors[5]	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers[5]	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H
MM0 through MM7[5]	Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium II and Pentium III Processors Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium with MMX Technology Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	Pwr up or Reset: 0H INIT: Unchanged	If CPUID.01H:SSE is 1 — Pwr up or Reset: 0H INIT: Unchanged	NA
MXCSR	Pwr up or Reset: 1F80H INIT: Unchanged	Pentium III processor only- Pwr up or Reset: 1F80H INIT: Unchanged	NA
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H

Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
DR7	00000400H	00000400H	00000400H
Time-Stamp Counter	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
Perf. Counters and Event Select	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
All Other MSRs	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged
Data and Code Cache, TLBs	Invalid[6]	Invalid[6]	Invalid[6]
Fixed MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Variable MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Machine-Check Architecture	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Not Implemented
APIC	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged
R8-R15[7]	0000000000000000H	0000000000000000H	N.A.
XMM8-XMM15[7]	Pwr up or Reset: 0H INIT: Unchanged	Pwr up or Reset: 0H INIT: Unchanged	N.A.

NOTES:

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where “n” is the Extended Model Value for the respective processor.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. If the processor supports IA-32e mode.

15. Updates to Chapter 25, Volume 3C

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

Changes to this chapter: Updated term "IA-32e paging" to "4-level paging"; included footnote on first usage of new term. Updates to section 25.5.6.1 "Convertible EPT Violations".

CHAPTER 25
VMX NON-ROOT OPERATION

In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment. This chapter describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation). The differences between VMX non-root operation and ordinary processor operation are described in the following sections:

- Section 25.1, "Instructions That Cause VM Exits"
- Section 25.2, "Other Causes of VM Exits"
- Section 25.3, "Changes to Instruction Behavior in VMX Non-Root Operation"
- Section 25.4, "Other Changes in VMX Non-Root Operation"
- Section 25.5, "Features Specific to VMX Non-Root Operation"
- Section 25.6, "Unrestricted Guests"

Chapter 26, "VM Entries," describes the data control structures that govern VMX non-root operation. Chapter 26, "VM Entries," describes the operation of VM entries by which the processor transitions from VMX root operation to VMX non-root operation. Chapter 25, "VMX Non-Root Operation," describes the operation of VM exits by which the processor transitions from VMX non-root operation to VMX root operation.

Chapter 28, "VMX Support for Address Translation," describes two features that support address translation in VMX non-root operation. Chapter 29, "APIC Virtualization and Virtual Interrupts," describes features that support virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC) in VMX non-root operation.

25.1 INSTRUCTIONS THAT CAUSE VM EXITS

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are "fault-like," meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction. Section 27.1 details architectural state in the context of a VM exit.

Section 25.1.1 defines the prioritization between faults and VM exits for instructions subject to both. Section 25.1.2 identifies instructions that cause VM exits whenever they are executed in VMX non-root operation (and thus can never be executed in VMX non-root operation). Section 25.1.3 identifies instructions that cause VM exits depending on the settings of certain VM-execution control fields (see Section 24.6).

25.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level,[1] and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.[2]
- Faults incurred while fetching instruction operands have priority over VM exits that are conditioned based on the contents of those operands (see LMSW in Section 25.1.3).
- VM exits caused by execution of the INS and OUTS instructions (resulting either because the "unconditional I/O exiting" VM-execution control is 1 or because the "use I/O bitmaps control is 1) have priority over the following faults:

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.
2. MOV DR is an exception to this rule; see Section 25.1.3.

- A general-protection fault due to the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) being unusable
- A general-protection fault due to an offset beyond the limit of the relevant segment
- An alignment-check exception

- Fault-like VM exits have priority over exceptions other than those mentioned above. For example, RDMSR of a non-existent MSR with CPL = 0 generates a VM exit and not a general-protection exception.

When Section 25.1.2 or Section 25.1.3 (below) identify an instruction execution that may lead to a VM exit, it is assumed that the instruction does not incur a fault that takes priority over a VM exit.

25.1.2 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC,[1] INVD, and XSETBV. This is also true of instructions introduced with VMX, which include: INVEPT, INVVPID, VMCALL,[2] VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

25.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:[3]

- **CLTS**. The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **ENCLS**. The ENCLS instruction causes a VM exit if the "enable ENCLS exiting" VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLS-exiting bitmap is 1 (see Section 24.6.16).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLS-exiting bitmap is 1.
- **HLT**. The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- **IN**, **INS/INSB/INSW/INSD**, **OUT**, **OUTS/OUTSB/OUTSW/OUTSD**. The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:
  - If both controls are 0, the instruction executes normally.
  - If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.
  - If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 24.6.4). If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).

  See Section 25.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.
- **INVLPG**. The INVLPG instruction causes a VM exit if the "INVLPG exiting" VM-execution control is 1.
- **INVPCID**. The INVPCID instruction causes a VM exit if the "INVLPG exiting" and "enable INVPCID" VM-execution controls are both 1.

1. An execution of GETSEC in VMX non-root operation causes a VM exit if CR4.SMXE[Bit 14] = 1 regardless of the value of CPL or RAX. An execution of GETSEC causes an invalid-opcode exception (#UD) if CR4.SMXE[Bit 14] = 0.
2. Under the dual-monitor treatment of SMIs and SMM, executions of VMCALL cause SMM VM exits in VMX root operation outside SMM. See Section 34.15.2.
3. Many of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

- **LGDT**, **LIDT**, **LLDT**, **LTR**, **SGDT**, **SIDT**, **SLDT**, **STR**. These instructions cause VM exits if the "descriptor-table exiting" VM-execution control is 1.
- **LMSW**. In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  - — The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - — For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
- **MONITOR**. The MONITOR instruction causes a VM exit if the "MONITOR exiting" VM-execution control is 1.
- **MOV from CR3**. The MOV from CR3 instruction causes a VM exit if the "CR3-store exiting" VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
- **MOV from CR8**. The MOV from CR8 instruction causes a VM exit if the "CR8-store exiting" VM-execution control is 1.
- **MOV to CR0**. The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
- **MOV to CR3**. The MOV to CR3 instruction causes a VM exit unless the "CR3-load exiting" VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Only the first *n* CR3-target values are considered, where *n* is the CR3-target count. If the "CR3-load exiting" VM-execution control is 1 and the CR3-target count is 0, MOV to CR3 always causes a VM exit.

  The first processors to support the virtual-machine extensions supported only the 1-setting of the "CR3-load exiting" VM-execution control. These processors always consult the CR3-target controls to determine whether an execution of MOV to CR3 causes a VM exit.
- **MOV to CR4**. The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- **MOV to CR8**. The MOV to CR8 instruction causes a VM exit if the "CR8-load exiting" VM-execution control is 1.
- **MOV DR**. The MOV DR instruction causes a VM exit if the "MOV-DR exiting" VM-execution control is 1. Such VM exits represent an exception to the principles identified in Section 25.1.1 in that they take priority over the following: general-protection exceptions based on privilege level; and invalid-opcode exceptions that occur because CR4.DE=1 and the instruction specified access to DR4 or DR5.
- **MWAIT**. The MWAIT instruction causes a VM exit if the "MWAIT exiting" VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 25.3).
- **PAUSE**. The behavior of each of this instruction depends on CPL and the settings of the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls:
  - — CPL = 0.
    - If the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls are both 0, the PAUSE instruction executes normally.
    - If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit (the "PAUSE-loop exiting" VM-execution control is ignored if CPL = 0 and the "PAUSE exiting" VM-execution control is 1).
    - If the "PAUSE exiting" VM-execution control is 0 and the "PAUSE-loop exiting" VM-execution control is 1, the following treatment applies.

      The processor determines the amount of time between this execution of PAUSE and the previous execution of PAUSE at CPL 0. If this amount of time exceeds the value of the VM-execution control field PLE_Gap, the processor considers this execution to be the first execution of PAUSE in a loop. (It also does so for the first execution of PAUSE at CPL 0 after VM entry.)

      Otherwise, the processor determines the amount of time since the most recent execution of PAUSE that was considered to be the first in a loop. If this amount of time exceeds the value of the VM-execution control field PLE_Window, a VM exit occurs.

For purposes of these computations, time is measured based on a counter that runs at the same rate as the timestamp counter (TSC).

— CPL > 0.

- If the “PAUSE exiting” VM-execution control is 0, the PAUSE instruction executes normally.
- If the “PAUSE exiting” VM-execution control is 1, the PAUSE instruction causes a VM exit.

The “PAUSE-loop exiting” VM-execution control is ignored if CPL > 0.

- **RDMSR**. The RDMSR instruction causes a VM exit if any of the following are true:
  — The “use MSR bitmaps” VM-execution control is 0.
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  — The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in read bitmap for low MSRs is 1, where *n* is the value of ECX.
  — The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in read bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

  See Section 24.6.9 for details regarding how these bitmaps are identified.
- **RDPMC**. The RDPMC instruction causes a VM exit if the “RDPMC exiting” VM-execution control is 1.
- **RDRAND**. The RDRAND instruction causes a VM exit if the “RDRAND exiting” VM-execution control is 1.
- **RDSEED**. The RDSEED instruction causes a VM exit if the “RDSEED exiting” VM-execution control is 1.
- **RDTSC**. The RDTSC instruction causes a VM exit if the “RDTSC exiting” VM-execution control is 1.
- **RDTSCP**. The RDTSCP instruction causes a VM exit if the “RDTSC exiting” and “enable RDTSCP” VM-execution controls are both 1.
- **RSM**. The RSM instruction causes a VM exit if executed in system-management mode (SMM).[1]
- **VMREAD**. The VMREAD instruction causes a VM exit if any of the following are true:
  — The “VMCS shadowing” VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit *n* in VMREAD bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMREAD bitmap is identified.

  If the VMREAD instruction does not cause a VM exit, it reads from the VMCS referenced by the VMCS link pointer. See Chapter 30, “VMREAD—Read Field from Virtual-Machine Control Structure” for details of the operation of the VMREAD instruction.
- **VMWRITE**. The VMWRITE instruction causes a VM exit if any of the following are true:
  — The “VMCS shadowing” VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit *n* in VMWRITE bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMWRITE bitmap is identified.

  If the VMWRITE instruction does not cause a VM exit, it writes to the VMCS referenced by the VMCS link pointer. See Chapter 30, “VMWRITE—Write Field to Virtual-Machine Control Structure” for details of the operation of the VMWRITE instruction.
- **WBINVD**. The WBINVD instruction causes a VM exit if the “WBINVD exiting” VM-execution control is 1.
- **WRMSR**. The WRMSR instruction causes a VM exit if any of the following are true:
  — The “use MSR bitmaps” VM-execution control is 0.
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  — The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in write bitmap for low MSRs is 1, where *n* is the value of ECX.

1. Execution of the RSM instruction outside SMM causes an invalid-opcode exception regardless of whether the processor is in VMX operation. It also does so in VMX root operation in SMM; see Section 34.15.3.

— The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in write bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

See Section 24.6.9 for details regarding how these bitmaps are identified.

- **XRSTORS**. The XRSTORS instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).
- **XSAVES**. The XSAVES instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).

25.2 OTHER CAUSES OF VM EXITS

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

- **Exceptions**. Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 24.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT3, INTO, BOUND, and UD.

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

  Thus, if software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If software desires VM exits on no page faults, it can set bit 14 in the exception bitmap to 1, the page-fault error-code mask field to 00000000H, and the page-fault error-code match field to FFFFFFFFH.
- **Triple fault**. A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.
- **External interrupts**. An external interrupt causes a VM exit if the "external-interrupt exiting" VM-execution control is 1. (See Section 25.6 for an exception.) Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)
- **Non-maskable interrupts (NMIs)**. An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
- **INIT signals**. INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)
- **Start-up IPIs (SIPIs)**. **SIPIs cause VM exits**. If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
- **Task switches**. Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. See Section 25.4.2.
- **System-management interrupts (SMIs)**. If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 34.15.2.[1]

- **VMX-preemption timer**. A VM exit occurs when the timer counts down to zero. See Section 25.5.1 for details of operation of the VMX-preemption timer.

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

  These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

- If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS (see Table 24-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.5).

  Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.
- If the "NMI-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS (see Table 24-3). (A logical processor may also prevent such a VM exit if there is blocking of events by STI.) Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.6).

  VM exits caused by the VMX-preemption timer and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

25.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:[1]

- **CLTS**. Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 23.8), in which case CLTS causes a general-protection exception.
  - If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.
- **INVPCID**. Behavior of the INVPCID instruction is determined first by the setting of the "enable INVPCID" VM-execution control:
  - If the "enable INVPCID" VM-execution control is 0, INVPCID causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.

1. Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 34.14.1.

1. Some of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

— If the "enable INVPCID" VM-execution control is 1, treatment is based on the setting of the "INVLPG exiting" VM-execution control:
  - If the "INVLPG exiting" VM-execution control is 0, INVPCID operates normally.
  - If the "INVLPG exiting" VM-execution control is 1, INVPCID causes a VM exit.

- **IRET**. Behavior of IRET with regard to NMI blocking (see Table 24-3) is determined by the settings of the "NMI exiting" and "virtual NMIs" VM-execution controls:

  — If the "NMI exiting" VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0; see Section 26.2.1.1.)

  — If the "NMI exiting" VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the "virtual NMIs" VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

  The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.

- **LMSW**. Outside of VMX non-root operation, LMSW loads its source operand into CR0[3:0], but it does not clear CR0.PE if that bit is set. In VMX non-root operation, an execution of LMSW that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0[3:0] corresponding to a bit set in the CR0 guest/host mask. An attempt to set any other bit in CR0[3:0] to a value not supported in VMX operation (see Section 23.8) causes a general-protection exception. Attempts to clear CR0.PE are ignored without fault.

- **MOV from CR0**. The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

- **MOV from CR3**. If the "enable EPT" VM-execution control is 1 and an execution of MOV from CR3 does not cause a VM exit (see Section 25.1.3), the value loaded from CR3 is a guest-physical address; see Section 28.2.1.

- **MOV from CR4**. The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow.

  Depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.

- **MOV from CR8**. If the MOV from CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 29.3.

- **MOV to CR0**. An execution of MOV to CR0 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. Treatment of attempts to modify other bits in CR0 depends on the setting of the "unrestricted guest" VM-execution control:

  — If the control is 0, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 to a value not supported in VMX operation (see Section 23.8).

  — If the control is 1, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 other than bit 0 (PE) or bit 31 (PG) to a value not supported in VMX operation. It remains the case, however, that MOV to CR0 causes a general-protection exception if it would result in CR0.PE = 0 and CR0.PG = 1 or if it would result in CR0.PG = 1, CR4.PAE = 0, and IA32_EFER.LME = 1.

- **MOV to CR3**. If the "enable EPT" VM-execution control is 1 and an execution of MOV to CR3 does not cause a VM exit (see Section 25.1.3), the value loaded into CR3 is treated as a guest-physical address; see Section 28.2.1.

  — If PAE paging is not being used, the instruction does not use the guest-physical address to access memory and it does not cause it to be translated through EPT.[1]

— If PAE paging is being used, the instruction translates the guest-physical address through EPT and uses the result to load the four (4) page-directory-pointer-table entries (PDPTEs). The instruction does not use the guest-physical addresses the PDPTEs to access memory and it does not cause them to be translated through EPT.

- **MOV to CR4**. An execution of MOV to CR4 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit in CR4 (not corresponding to a bit set in the CR4 guest/host mask) to a value not supported in VMX operation (see Section 23.8).
- **MOV to CR8**. If the MOV to CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 29.3.
- **MWAIT**. Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the "MWAIT exiting" VM-execution control:

— If the "MWAIT exiting" VM-execution control is 1, MWAIT causes a VM exit.

— If the "MWAIT exiting" VM-execution control is 0, MWAIT operates normally if one of the following are true: (1) ECX[0] is 0; (2) RFLAGS.IF = 1; or both of the following are true: (a) the "interrupt-window exiting" VM-execution control is 0; and (b) the logical processor has not recognized a pending virtual interrupt (see Section 29.2.1).

— If the "MWAIT exiting" VM-execution control is 0, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state if either the "interrupt-window exiting" VM-execution control is 1 or the logical processor has recognized a pending virtual interrupt; instead, control passes to the instruction following the MWAIT instruction.

- **RDMSR**. Section 25.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:

— If ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the instruction is determined by the setting of the "use TSC offsetting" VM-execution control:

  - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
  - If the control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

    — If the control is 0, RDMSR loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.

    — If the control is 1, RDMSR first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

  The 1-setting of the "use TSC-offsetting" VM-execution control does not affect executions of RDMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.

— If ECX is in the range 800H–8FFH (indicating an APIC MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 29.5.

- **RDPID**. Behavior of the RDPID instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:

— If the "enable RDTSCP" VM-execution control is 0, RDPID causes an invalid-opcode exception (#UD).

— If the "enable RDTSCP" VM-execution control is 1, RDPID operates normally.

- **RDTSC**. Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

— If both controls are 0, RDTSC operates normally.

— If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

  - If the control is 0, RDTSC loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
  - If the control is 1, RDTSC first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

— If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit.

- **RDTSCP**. Behavior of the RDTSCP instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:

  — If the "enable RDTSCP" VM-execution control is 0, RDTSCP causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.

  — If the "enable RDTSCP" VM-execution control is 1, treatment is based on the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

    - If both controls are 0, RDTSCP operates normally.
    - If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

      — If the control is 0, RDTSCP loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.

      — If the control is 1, RDTSCP first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

      In either case, RDTSCP also loads ECX with the value of bits 31:0 of the IA32_TSC_AUX MSR.

    - If the "RDTSC exiting" VM-execution control is 1, RDTSCP causes a VM exit.

- **SMSW**. The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

- **WRMSR**. Section 25.1.3 identifies when executions of the WRMSR instruction cause VM exits. If such an execution neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:

  — If ECX contains 79H (indicating IA32_BIOS_UPDT_TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.

  — On processors that support Intel PT but which do not allow it to be used in VMX operation, if ECX contains 570H (indicating the IA32_RTIT_CTL MSR), the instruction causes a general-protection exception if it attempts to set IA32_RTIT_CTL.TraceEn.[1]

1. Software should read the VMX capability MSR IA32_VMX_MISC to determine whether the processor allows Intel PT to be used in VMX operation (see Appendix A.6).

— If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), or 83FH (self-IPI MSR), instruction behavior may modified if the “virtualize x2APIC mode” VM-execution control is 1; see Section 29.5.

- **XRSTORS**. Behavior of the XRSTORS instruction is determined first by the setting of the “enable XSAVES/XRSTORS” VM-execution control:
  — If the “enable XSAVES/XRSTORS” VM-execution control is 0, XRSTORS causes an invalid-opcode exception (#UD).
  — If the “enable XSAVES/XRSTORS” VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XRSTORS causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XRSTORS operates normally.
- **XSAVES**. Behavior of the XSAVES instruction is determined first by the setting of the “enable XSAVES/XRSTORS” VM-execution control:
  — If the “enable XSAVES/XRSTORS” VM-execution control is 0, XSAVES causes an invalid-opcode exception (#UD).
  — If the “enable XSAVES/XRSTORS” VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XSAVES causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XSAVES operates normally.

25.4 OTHER CHANGES IN VMX NON-ROOT OPERATION

Treatments of event blocking and of task switches differ in VMX non-root operation as described in the following sections.

25.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the “external-interrupt exiting” VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0).
- If the “external-interrupt exiting” VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).
- If the “NMI exiting” VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).

25.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:
   a. If CALL, INT *n*, or JMP accesses a task gate in IA-32e mode, a general-protection exception occurs.
   b. If CALL, INT *n*, INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.

c. If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general-protection exception occurs.

d. If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.

e. If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.

f. If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.

g. If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs.

h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.

2. Checks are made on the new TSS selector (for example, that is within GDT limits).
3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).
4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.

Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

If an attempt at a task switch through a task gate in the IDT causes an exception (before generating a VM exit due to the task switch) and that exception causes a VM exit, information about the event whose delivery that accessed the task gate is recorded in the IDT-vectoring information fields and information about the exception that caused the VM exit is recorded in the VM-exit interruption-information fields. See Section 27.2. The fact that a task gate was being accessed is not recorded in the VMCS.

If an attempt at a task switch through a task gate in the IDT causes VM exit due to the task switch, information about the event whose delivery accessed the task gate is recorded in the IDT-vectoring fields of the VMCS. Since the cause of such a VM exit is a task switch and not an interruption, the valid bit for the VM-exit interruption information field is 0. See Section 27.2.

25.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION

Some VM-execution controls support features that are specific to VMX non-root operation. These are the VMX-preemption timer (Section 25.5.1) and the monitor trap flag (Section 25.5.2), translation of guest-physical addresses (Section 25.5.3), VM functions (Section 25.5.5), and virtualization exceptions (Section 25.5.6).

25.5.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of "activate VMX-preemption timer" VM-execution control, the **VMX-preemption timer** counts down (from the value loaded by VM entry; see Section 26.6.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 25.2).

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

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

Treatment of the timer in the case of system management interrupts (SMIs) and system-management mode (SMM) depends on whether the treatment of SMIs and SMM:

- If the default treatment of SMIs and SMM (see Section 34.14) is active, the VMX-preemption timer counts across an SMI to VMX non-root operation, subsequent execution in SMM, and the return from SMM via the RSM instruction. However, the timer can cause a VM exit only from VMX non-root operation. If the timer expires during SMI, in SMM, or during RSM, a timer-induced VM exit occurs immediately after RSM with its normal priority unless it is blocked based on activity state (Section 25.2).
- If the dual-monitor treatment of SMIs and SMM (see Section 34.15) is active, transitions into and out of SMM are VM exits and VM entries, respectively. The treatment of the VMX-preemption timer by those transitions is mostly the same as for ordinary VM exits and VM entries; Section 34.15.2 and Section 34.15.4 detail some differences.

25.5.2 Monitor Trap Flag

The **monitor trap flag** is a debugging feature that causes VM exits to occur on certain instruction boundaries in VMX non-root operation. Such VM exits are called **MTF VM exits**. An MTF VM exit may occur on an instruction boundary in VMX non-root operation as follows:

- If the "monitor trap flag" VM-execution control is 1 and VM entry is injecting a vectored event (see Section 26.5.1), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry.
- If VM entry is injecting a pending MTF VM exit (see Section 26.5.2), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry. This is the case even if the "monitor trap flag" VM-execution control is 0.
- If the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and a pending event (e.g., debug exception or interrupt) is delivered before an instruction can execute, an MTF VM exit is pending on the instruction boundary following delivery of the event (or any nested exception).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is a REP-prefixed string instruction:
  - If the first iteration of the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the first iteration of the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary after that iteration.
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is the XBEGIN instruction. In this case, an MTF VM exit is pending at the fallback instruction address of the XBEGIN instruction. This behavior applies regardless of whether advanced debugging of RTM transactional regions has been enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is neither a REP-prefixed string instruction or the XBEGIN instruction:
  - If the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).[1]
  - If the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary following execution of that instruction. If the instruction is INT3 or INTO, this boundary follows delivery of any software exception. If the instruction is INT *n*, this boundary follows delivery of a software interrupt. If the instruction is HLT, the MTF VM exit will be from the HLT activity state.

No MTF VM exit occurs if another VM exit occurs before reaching the instruction boundary on which an MTF VM exit would be pending (e.g., due to an exception or triple fault).

An MTF VM exit occurs on the instruction boundary on which it is pending unless a higher priority event takes precedence or the MTF VM exit is blocked due to the activity state:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over MTF VM exits. MTF VM exits take priority over debug-trap exceptions and lower priority events.

1. This item includes the cases of an invalid opcode exception—#UD— generated by the UD instruction and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.

- No MTF VM exit occurs if the processor is in either the shutdown activity state or wait-for-SIPI activity state. If a non-maskable interrupt subsequently takes the logical processor out of the shutdown activity state without causing a VM exit, an MTF VM exit is pending after delivery of that interrupt.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 43.2 for details.

25.5.3 Translation of Guest-Physical Addresses Using EPT

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain physical addresses are treated as guest-physical addresses and are not used to access memory directly. Instead, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

Details of the EPT mechanism are given in Section 28.2.

25.5.4 APIC Virtualization

APIC virtualization is a collection of features that can be used to support the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC). When APIC virtualization is enabled, the processor emulates many accesses to the APIC, tracks the state of the virtual APIC, and delivers virtual interrupts — all in VMX non-root operation without a VM exit.

Details of the APIC virtualization are given in Chapter 29.

25.5.5 VM Functions

A **VM function** is an operation provided by the processor that can be invoked from VMX non-root operation without a VM exit. VM functions are enabled and configured by the settings of different fields in the VMCS. Software in VMX non-root operation invokes a VM function with the **VMFUNC** instruction; the value of EAX selects the specific VM function being invoked.

Section 25.5.5.1 explains how VM functions are enabled. Section 25.5.5.2 specifies the behavior of the VMFUNC instruction. Section 25.5.5.3 describes a specific VM function called **EPTP switching**.

25.5.5.1 Enabling VM Functions

Software enables VM functions generally by setting the “enable VM functions” VM-execution control. A specific VM function is enabled by setting the corresponding VM-function control.

Suppose, for example, that software wants to enable EPTP switching (VM function 0; see Section 24.6.14).To do so, it must set the “activate secondary controls” VM-execution control (bit 31 of the primary processor-based VM-execution controls), the “enable VM functions” VM-execution control (bit 13 of the secondary processor-based VM-execution controls) and the “EPTP switching” VM-function control (bit 0 of the VM-function controls).

25.5.5.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (#UD) if the “enable VM functions” VM-execution controls is 0[1] or the value of EAX is greater than 63 (only VM functions 0–63 can be enable). Otherwise, the instruction causes a VM exit if the bit at position EAX is 0 in the VM-function controls (the selected VM function is not enabled). If such a VM exit occurs, the basic exit reason used is 59 (3BH), indicating “VMFUNC”, and the length of the VMFUNC instruction is saved into the VM-exit instruction-length field. If the instruction causes neither an invalid-opcode exception nor a VM exit due to a disabled VM function, it performs the functionality of the VM function specified by the value in EAX.

---

1. “Enable VM functions” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “enable VM functions” VM-execution control were 0. See Section 24.6.2.

Individual VM functions may perform additional fault checking (e.g., one might cause a general-protection exception if CPL > 0). In addition, specific VM functions may include checks that might result in a VM exit. If such a VM exit occurs, VM-exit information is saved as described in the previous paragraph. The specification of a VM function may indicate that additional VM-exit information is provided.

The specific behavior of the EPTP-switching VM function (including checks that result in VM exits) is given in Section 25.5.5.3.

25.5.5.3 EPTP Switching

EPTP switching is VM function 0. This VM function allows software in VMX non-root operation to load a new value for the EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy (see Section 28.2 for details of the operation of EPT). Software is limited to selecting from a list of potential EPTP values configured in advance by software in VMX root operation.

Specifically, the value of ECX is used to select an entry from the EPTP list, the 4-KByte structure referenced by the EPTP-list address (see Section 24.6.14; because this structure contains 512 8-Byte entries, VMFUNC causes a VM exit if ECX ≥ 512). If the selected entry is a valid EPTP value (it would not cause VM entry to fail; see Section 26.2.1.1), it is stored in the EPTP field of the current VMCS and is used for subsequent accesses using guest-physical addresses. The following pseudocode provides details:

```
IF ECX ≥ 512
    THEN VM exit;
    ELSE
        tent_EPTP ← 8 bytes from EPTP-list address + 8 * ECX;
        IF tent_EPTP is not a valid EPTP value (would cause VM entry to fail if in EPTP)
            THEN VMexit;
            ELSE
                write tent_EPTP to the EPTP field in the current VMCS;
                use tent_EPTP as the new EPTP value for address translation;
                IF processor supports the 1-setting of the "EPT-violation #VE" VM-execution control
                    THEN
                        write ECX[15:0] to EPTP-index field in current VMCS;
                        use ECX[15:0] as EPTP index for subsequent EPT-violation virtualization exceptions (see Section 25.5.6.2);
                FI;
        FI;
FI;
```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

As noted in Section 25.5.5.2, an execution of the EPTP-switching VM function that causes a VM exit (as specified above), uses the basic exit reason 59, indicating "VMFUNC". The length of the VMFUNC instruction is saved into the VM-exit instruction-length field. No additional VM-exit information is provided.

An execution of VMFUNC loads EPTP from the EPTP list (and thus does not cause a fault or VM exit) is called an **EPTP-switching VMFUNC**. After an EPTP-switching VMFUNC, control passes to the next instruction. The logical processor starts creating and using guest-physical and combined mappings associated with the new value of bits 51:12 of EPTP; the combined mappings created and used are associated with the current VPID and PCID (these are not changed by VMFUNC).[1] If the "enable VPID" VM-execution control is 0, an EPTP-switching VMFUNC invalidates combined mappings associated with VPID 0000H (for all PCIDs and for all EP4TA values, where EP4TA is the value of bits 51:12 of EPTP).

Because an EPTP-switching VMFUNC may change the translation of guest-physical addresses, it may affect use of the guest-physical address in CR3. The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration due to the translation of that guest-physical address through the new EPT paging structures. The following items provide details that apply if CR0.PG = 1:

1. If the "enable VPID" VM-execution control is 0, the current VPID is 0000H; if CR4.PCIDE = 0, the current PCID is 000H.

- If 32-bit paging or 4-level paging[1] is in use (either CR4.PAE = 0 or IA32_EFER.LMA = 1), the next memory access with a linear address uses the translation of the guest-physical address in CR3 through the new EPT paging structures. As a result, this access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.
- If PAE paging is in use (CR4.PAE = 1 and IA32_EFER.LMA = 0), an EPTP-switching VMFUNC **does not** load the four page-directory-pointer-table entries (PDPTEs) from the guest-physical address in CR3. The logical processor continues to use the four guest-physical addresses already present in the PDPTEs. The guest-physical address in CR3 is not translated through the new EPT paging structures (until some operation that would load the PDPTEs).

  The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during the translation of a guest-physical address in any of the PDPTEs. A subsequent memory access with a linear address uses the translation of the guest-physical address in the appropriate PDPTE through the new EPT paging structures. As a result, such an access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.

If an EPTP-switching VMFUNC establishes an EPTP value that enables accessed and dirty flags for EPT (by setting bit 6), subsequent memory accesses may fail to set those flags as specified if there has been no appropriate execution of INVEPT since the last use of an EPTP value that does not enable accessed and dirty flags for EPT (because bit 6 is clear) and that is identical to the new value on bits 51:12.

IF the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control, an EPTP-switching VMFUNC loads the value in ECX[15:0] into to EPTP-index field in current VMCS. Subsequent EPT-violation virtualization exceptions will save this value into the virtualization-exception information area (see Section 25.5.6.2);

25.5.6 Virtualization Exceptions

A **virtualization exception** is a new processor exception. It uses vector 20 and is abbreviated #VE.

A virtualization exception can occur only in VMX non-root operation. Virtualization exceptions occur only with certain settings of certain VM-execution controls. Generally, these settings imply that certain conditions that would normally cause VM exits instead cause virtualization exceptions

In particular, the 1-setting of the “EPT-violation #VE” VM-execution control causes some EPT violations to generate virtualization exceptions instead of VM exits. Section 25.5.6.1 provides the details of how the processor determines whether an EPT violation causes a virtualization exception or a VM exit.

When the processor encounters a virtualization exception, it saves information about the exception to the virtualization-exception information area; see Section 25.5.6.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 25.5.6.3 for details.

25.5.6.1 Convertible EPT Violations

If the “EPT-violation #VE” VM-execution control is 0 (e.g., on processors that do not support this feature), EPT violations always cause VM exits. If instead the control is 1, certain EPT violations may be converted to cause virtualization exceptions instead; such EPT violations are **convertible**.

The values of certain EPT paging-structure entries determine which EPT violations are convertible. Specifically, bit 63 of certain EPT paging-structure entries may be defined to mean **suppress #VE**:

- If bits 2:0 of an EPT paging-structure entry are all 0, the entry is not **present**.[2] If the processor encounters such an entry while translating a guest-physical address, it causes an EPT violation. The EPT violation is convertible if and only if bit 63 of the entry is 0.
- If an EPT paging-structure entry is present, the following cases apply:

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.
2. If the “mode-based execute control for EPT” VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 **or bit 10** is 1.

— If the value of the EPT paging-structure entry is not supported, the entry is **misconfigured**. If the processor encounters such an entry while translating a guest-physical address, it causes an EPT misconfiguration (not an EPT violation). EPT misconfigurations always cause VM exits.

— If the value of the EPT paging-structure entry is supported, the following cases apply:

  - If bit 7 of the entry is 1, or if the entry is an EPT PTE, the entry maps a page. If the processor uses such an entry to translate a guest-physical address, and if an access to that address causes an EPT violation, the EPT violation is convertible if and only if bit 63 of the entry is 0.
  - If bit 7 of the entry is 0 and the entry is not an EPT PTE, the entry references another EPT paging structure. The processor does not use the value of bit 63 of the entry to determine whether any subsequent EPT violation is convertible.

If an access to a guest-physical address causes an EPT violation, bit 63 of exactly one of the EPT paging-structure entries used to translate that address is used to determine whether the EPT violation is convertible: either a entry that is not present (if the guest-physical address does not translate to a physical address) or an entry that maps a page (if it does).

A convertible EPT violation instead causes a virtualization exception if the following all hold:

- CR0.PE = 1;
- the logical processor is not in the process of delivering an event through the IDT; and
- the 32 bits at offset 4 in the virtualization-exception information area are all 0.

Delivery of virtualization exceptions writes the value FFFFFFFFH to offset 4 in the virtualization-exception information area (see Section 25.5.6.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

25.5.6.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 24.6.18). Table 25-1 enumerates the data saved and the format of the area.

Table 25-1. Format of the Virtualization-Exception Information Area

Byte Offset	Contents
0	The 32-bit value that would have been saved into the VMCS as an exit reason had a VM exit occurred instead of the virtualization exception. For EPT violations, this value is 48 (00000030H)
4	FFFFFFFFH
8	The 64-bit value that would have been saved into the VMCS as an exit qualification had a VM exit occurred instead of the virtualization exception
16	The 64-bit value that would have been saved into the VMCS as a guest-linear address had a VM exit occurred instead of the virtualization exception
24	The 64-bit value that would have been saved into the VMCS as a guest-physical address had a VM exit occurred instead of the virtualization exception
32	The current 16-bit value of the EPTP index VM-execution control (see Section 24.6.18 and Section 25.5.5.3)

25.5.6.3 Delivery of Virtualization Exceptions

After saving virtualization-exception information, the processor treats a virtualization exception as it does other exceptions:

- If bit 20 (#VE) is 1 in the exception bitmap in the VMCS, a virtualization exception causes a VM exit (see below). If the bit is 0, the virtualization exception is delivered using gate descriptor 20 in the IDT.

- Virtualization exceptions produce no error code. Delivery of a virtualization exception pushes no error code on the stack.
- With respect to double faults, virtualization exceptions have the same severity as page faults. If delivery of a virtualization exception encounters a nested fault that is either contributory or a page fault, a double fault (#DF) is generated. See Chapter 6, "Interrupt 8—Double Fault Exception (#DF)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

  It is not possible for a virtualization exception to be encountered while delivering another exception (see Section 25.5.6.1).

If a virtualization exception causes a VM exit directly (because bit 20 is 1 in the exception bitmap), information about the exception is saved normally in the VM-exit interruption information field in the VMCS (see Section 27.2.2). Specifically, the event is reported as a hardware exception with vector 20 and no error code. Bit 12 of the field (NMI unblocking due to IRET) is set normally.

If a virtualization exception causes a VM exit indirectly (because bit 20 is 0 in the exception bitmap and delivery of the exception generates an event that causes a VM exit), information about the exception is saved normally in the IDT-vectoring information field in the VMCS (see Section 27.2.3). Specifically, the event is reported as a hardware exception with vector 20 and no error code.

25.6 UNRESTRICTED GUESTS

The first processors to support VMX operation require CR0.PE and CR0.PG to be 1 in VMX operation (see Section 23.8). This restriction implies that guest software cannot be run in unpaged protected mode or in real-address mode. Later processors support a VM-execution control called "unrestricted guest".[1] If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation. Such processors allow guest software to run in unpaged protected mode or in real-address mode. The following items describe the behavior of such software:

- The MOV CR0 instructions does not cause a general-protection exception simply because it would set either CR0.PE and CR0.PG to 0. See Section 25.3 for details.
- A logical processor treats the values of CR0.PE and CR0.PG in VMX non-root operation just as it does outside VMX operation. Thus, if CR0.PE = 0, the processor operates as it does normally in real-address mode (for example, it uses the 16-bit interrupt table to deliver interrupts and exceptions). If CR0.PG = 0, the processor operates as it does normally when paging is disabled.
- Processor operation is modified by the fact that the processor is in VMX non-root operation and by the settings of the VM-execution controls just as it is in protected mode or when paging is enabled. Instructions, interrupts, and exceptions that cause VM exits in protected mode or when paging is enabled also do so in real-address mode or when paging is disabled. The following examples should be noted:
  - If CR0.PG = 0, page faults do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, invalid-TSS exceptions do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, the following instructions cause invalid-opcode exceptions and do not cause VM exits: INVEPT, INVVPID, LLDT, LTR, SLDT, STR, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.
- If CR0.PG = 0, each linear address is passed directly to the EPT mechanism for translation to a physical address.[2] The guest memory type passed on to the EPT mechanism is WB (writeback).

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "unrestricted guest" VM-execution control were 0. See Section 24.6.2.
2. As noted in Section 26.2.1.1, the "enable EPT" VM-execution control must be 1 if the "unrestricted guest" VM-execution control is 1.

16. Updates to Chapter 30, Volume 3C

Change bars show changes to Chapter 30 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to this chapter: Updates to the following instructions: VMFUNC, VMPTRLD, VMPTRST, VMREAD and VMWRITE.

VMFUNC—Invoke VM function

Opcode	Instruction	Description
NP 0F 01 D4	VMFUNC	Invoke VM function specified in EAX.

Description

This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. The value of EAX selects the specific VM function being invoked.

The behavior of each VM function (including any additional fault checking) is specified in Section 25.5.5, “VM Functions”.

Operation

```
Perform functionality of the VM function specified in EAX;
```

Flags Affected

Depends on the VM function specified in EAX. See Section 25.5.5, “VM Functions”.

Protected Mode Exceptions (not including those defined by specific VM functions)

#UD If executed outside VMX non-root operation.
If “enable VM functions” VM-execution control is 0.
If EAX ≥ 64.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
NP 0F C7 /6	VMPTRLD m64	Loads the current VMCS pointer from memory.

Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.[1]

The operand of this instruction is always 64 bits and is always in memory.

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        addr ← contents of 64-bit in-memory source operand;
        IF addr is not 4KB-aligned OR
        addr sets any bits beyond the physical-address width2
            THEN VMfail(VMPTRLD with invalid physical address);
        ELSIF addr = VMXON pointer
            THEN VMfail(VMPTRLD with VMXON pointer);
            ELSE
                rev ← 32 bits located at physical address addr;
                IF rev[30:0] ≠ VMCS revision identifier supported by processor OR
                rev[31] = 1 AND processor does not support 1-setting of “VMCS shadowing”
                    THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);
                    ELSE
                        current-VMCS pointer ← addr;
                        VMsucceed;
                FI;
        FI;
FI;
```

Flags Affected

See the operation section and Section 30.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.

---

1. Software should consult the VMX capability MSR VMX_BASIC to discover the VMCS revision identifier supported by this processor (see Appendix A, “VMX Capability Reporting Facility”).
2. If IA32_VMX_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix A.1.

	If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	If the memory source operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If operand is a register.
	If not in VMX operation.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	If the source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If operand is a register.
	If not in VMX operation.

VMPTRST—Store Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
NP 0F C7 /7	VMPTRST m64	Stores the current VMCS pointer into memory.

Description

Stores the current-VMCS pointer into a specified memory address. The operand of this instruction is always 64 bits and is always in memory.

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        64-bit in-memory destination operand ← current-VMCS pointer;
        VMsucceed;
FI;
```

Flags Affected

See the operation section and Section 30.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the destination operand is located in a read-only data segment or any code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory destination operand.
#SS(0)	If the memory destination operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If operand is a register.
	If not in VMX operation.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing the memory destination operand.
#SS(0)	If the destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If operand is a register.
	If not in VMX operation.

VMREAD—Read Field from Virtual-Machine Control Structure

Opcode	Instruction	Description
NP 0F 78	VMREAD r/m64, r64	Reads a specified VMCS field (in 64-bit mode).
NP 0F 78	VMREAD r/m32, r32	Reads a specified VMCS field (outside 64-bit mode).

Description

Reads a specified field from a VMCS and stores it into a specified destination operand (register or memory). In VMX root operation, the instruction reads from the current VMCS. If executed in VMX non-root operation, the instruction reads from the VMCS referenced by the VMCS link pointer field in the current VMCS.

The VMCS field is specified by the VMCS-field encoding contained in the register source operand. Outside IA-32e mode, the source operand has 32 bits, regardless of the value of CS.D. In 64-bit mode, the source operand has 64 bits.

The effective size of the destination operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the source operand is shorter than this effective operand size, the high bits of the destination operand are cleared to 0. If the VMCS field is longer, then the high bits of the field are not read.

Note that any faults resulting from accessing a memory destination operand can occur only after determining, in the operation section below, that the relevant VMCS pointer is valid and that the specified VMCS field is supported.

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation AND (“VMCS shadowing” is 0 OR source operand sets bits in range 63:15 OR
VMREAD bit corresponding to bits 14:0 of source operand is 1)[1]
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF (in VMX root operation AND current-VMCS pointer is not valid) OR
(in VMX non-root operation AND VMCS link pointer is not valid)
    THEN VMfailInvalid;
ELSIF source operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
    ELSE
        IF in VMX root operation
            THEN destination operand ← contents of field indexed by source operand in current VMCS;
            ELSE destination operand ← contents of field indexed by source operand in VMCS referenced by VMCS link pointer;
        FI;
        VMsucceed;
FI;
```

Flags Affected

See the operation section and Section 30.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

1. The VMREAD bit for a source operand is defined as follows. Let *x* be the value of bits 14:0 of the source operand and let *addr* be the VMREAD-bitmap address. The corresponding VMREAD bit is in bit position *x* & 7 of the byte at physical address *addr* | (x » 3).

	If a memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the destination operand is located in a read-only data segment or any code segment.
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	If a memory destination operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If not in VMX operation.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.

VMWRITE—Write Field to Virtual-Machine Control Structure

Opcode	Instruction	Description
NP 0F 79	VMWRITE r64, r/m64	Writes a specified VMCS field (in 64-bit mode)
NP 0F 79	VMWRITE r32, r/m32	Writes a specified VMCS field (outside 64-bit mode)

Description

Writes the contents of a primary source operand (register or memory) to a specified field in a VMCS. In VMX root operation, the instruction writes to the current VMCS. If executed in VMX non-root operation, the instruction writes to the VMCS referenced by the VMCS link pointer field in the current VMCS.

The VMCS field is specified by the VMCS-field encoding contained in the register secondary source operand. Outside IA-32e mode, the secondary source operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode, the secondary source operand has 64 bits.

The effective size of the primary source operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the secondary source operand is shorter than this effective operand size, the high bits of the primary source operand are ignored. If the VMCS field is longer, then the high bits of the field are cleared to 0.

Note that any faults resulting from accessing a memory source operand occur after determining, in the operation section below, that the relevant VMCS pointer is valid but before determining if the destination VMCS field is supported.

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation AND ("VMCS shadowing" is 0 OR secondary source operand sets bits in range 63:15 OR
VMWRITE bit corresponding to bits 14:0 of secondary source operand is 1)[1]
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF (in VMX root operation AND current-VMCS pointer is not valid) OR
(in VMX non-root operation AND VMCS-link pointer is not valid)
    THEN VMfailInvalid;
ELSIF secondary source operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSIF VMCS field indexed by secondary source operand is a VM-exit information field AND
processor does not support writing to such fields[2]
    THEN VMfailValid(VMWRITE to read-only VMCS component);
    ELSE
        IF in VMX root operation
            THEN field indexed by secondary source operand in current VMCS ← primary source operand;
            ELSE field indexed by secondary source operand in VMCS referenced by VMCS link pointer ← primary source operand;
        FI;
        VMsucceed;
FI;
```

1. The VMWRITE bit for a secondary source operand is defined as follows. Let *x* be the value of bits 14:0 of the secondary source operand and let *addr* be the VMWRITE-bitmap address. The corresponding VMWRITE bit is in bit position *x* & 7 of the byte at physical address *addr* | (x » 3).
2. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

Flags Affected

See the operation section and Section 30.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	If a memory source operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If not in VMX operation.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	If the memory source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.

17. Updates to Chapter 35, Volume 3C

Change bars show changes to Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Change to chapter: Updated term "IA-32e paging" to "4-level paging"; included footnote on first usage of new term. Updates to section 35.2.6.2 "Table of Physical Addresses (ToPA)".

Note: This chapter is now the chapter covering Intel® Processor Trace. In previous versions of this manual, chapter 35 covered model specific registers. The model specific registers have now moved to volume 4.

CHAPTER 35
INTEL® PROCESSOR TRACE

35.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The initial implementations of Intel PT offer **control flow tracing**, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

35.1.1 Features and Capabilities

Intel PT's control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

Intel PT can also be configured to log software-generated packets using PTWRITE, and packets describing processor power management events.

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32_RTIT_*. The capability provided by these configuration MSRs are enumerated by CPUID, see Section 35.3. Details of the MSRs for configuring Intel PT are described in Section 35.2.7.

35.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following categories of packets (for more details on the packets, see Section 35.4):

- Packets about basic information on program execution: These include:
  - Packet Stream Boundary (**PSB**) packets: PSB packets act as 'heartbeats' that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
  - Paging Information Packet (**PIP**): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
  - Time-Stamp Counter (**TSC**) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
  - Core Bus Ratio (**CBR**) packets: CBR packets contain the core:bus clock ratio.

— Overflow (**OVF**) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.

- Packets about control flow information:
  — Taken Not-Taken (**TNT**) packets: TNT packets track the "direction" of direct conditional branches (taken or not taken).
  — Target IP (**TIP**) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 35.4.2.2.
  — Flow Update Packets (**FUP**): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
  — **MODE** packets: These packets provide the decoder with important processor execution information so that it can properly interpret the dis-assembled binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).
- Packets inserted by software:
  — PTWRITE (PTW) packets: includes the value of the operand passed to the PTWRITE instruction (see "PTWRITE - Write Data to a Processor Trace Packet" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).
- Packets about processor power management events:
  — MWAIT packets: Indicate successful completion of an MWAIT operation to a C-state deeper than C0.0.
  — Power State Entry (PWRE) packets: Indicate entry to a C-state deeper than C0.0.
  — Power State Exit (PWRX) packets: Indicate exit from a C-state deeper than C0.0, returning to C0.
  — Execution Stopped (EXSTOP) packets: Indicate that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling.

35.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

35.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). There are three categories of COFI:

- Direct transfer COFI.
- Indirect transfer COFI.
- Far transfer COFI.

The following subsections describe the COFI events that result in trace packet generation. Table 35-1 lists branch instruction by COFI types. For detailed description of specific instructions, see *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

35.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two subcategories:

- **Conditional Branch (Jcc, J*CXZ) and LOOP**

  To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction.

  Jcc, J*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**

  There is no trace output required for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet, though TIP.PGD and TIP.PGE packets can be generated by unconditional direct jumps that toggle Intel PT enables (see Section 35.2.5).

35.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 35.3.1.1).

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**

  As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**

  When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

  A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

  — **RET Compression**

    A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with “corresponding” defined

as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be "compressed". In this case, only a single TNT bit of "taken" is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed in a given logical processor. For details, see "Indirect Transfer Compression for Returns (RET)" in Section 35.4.2.2.

35.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are "far transfers". This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 35-23 indicates exactly which IP will be included in the FUP generated by a far transfer.

35.2.2 Software Trace Instrumentation with PTWRITE

PTWRITE provides a mechanism by which software can instrument the Intel PT trace. PTWRITE is a ring3-accessible instruction that can be passed a register or memory variable, see "PTWRITE - Write Data to a Processor Trace Packet" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B* for details. The contents of that variable will be used as the payload for the PTW packet (see Table 35-40 "PTW Packet Definition"), inserted at the time of PTWRITE retirement, assuming PTWRITE is enabled and all other filtering conditions are met. Decode and analysis software will then be able to determine the meaning of the PTWRITE packet based on the IP of the associated PTWRITE instruction.

PTWRITE is enabled via IA32_RTIT_CTL.PTWEn[12] (see Table 35-6). Optionally, the user can use IA32_RTIT_CTL.FUPonPTW[5] to enable PTW packets to be followed by FUP packets containing the IP of the associated PTWRITE instruction.

35.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

— Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.

— The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.

— The last and deepest core C-state achieved during a sleep session.

— The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32_RTIT_CTL.PwrEvtEn[4].

35.2.4 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

35.2.4.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 35.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 35.2.5.1 for details.

35.2.4.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 35.3.5, “Context Switch Consideration”).

To trace for only a single CR3 value, software can write that value to the IA32_RTIT_CR3_MATCH MSR, and set IA32_RTIT_CTL.CR3Filter. When CR3 value does not match IA32_RTIT_CR3_MATCH and IA32_RTIT_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 35.2.5.3 for details. When CR3 does match IA32_RTIT_CR3_MATCH (or when IA32_RTIT_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32_RTIT_CTL.OS = 1). If not, no PIP packets will be seen.

35.2.4.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1. Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn_CFG fields in the IA32_RTIT_CTL MSR (Section 35.2.7.2), where the digit 'n' is a zero-based number that selects which address range is being configured. Each ADDRn_CFG field configures the use of the register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B (Section 35.2.7.5). IA32_RTIT_ADDRn_A defines the base and IA32_RTIT_ADDRn_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32_RTIT_ADDRn_A. IA32_RTIT_ADDRn_B]. There can be multiple such ranges, software can query CPUID (Section 35.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn_CFG is set to enable IP filtering (see Section 35.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 35.2.5.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 35.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 35.3.1).

Note that some packets, such as MTC (Section 35.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 35.4.1.

TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32_RTIT_CTL.ADDRn_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 35.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.

To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32_RTIT_CTL.TraceEn before the IA32_RTIT_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32_RTIT_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 35.2.6.2.

IP Filtering Example

The following table gives an example of IP filtering behavior. Assume that IA32_RTIT_ADDRn_A = the IP of RangeBase, and that IA32_RTIT_ADDRn_B = the IP of RangeLimit, while IA32_RTIT_CTL.ADDRn_CFG = 0x1 (enable ADDRn range as a FilterEn range).

Table 35-2. IP Filtering Packet Example

Code Flow	Packets

`Bar:`
`jmp RangeBase // jump into filter range`
`RangeBase:`
`jcc Foo // not taken`
`add eax, 1`
`Foo:`
`jmp RangeLimit+1 // jump out of filter range`
`RangeLimit:`
`nop`
`jcc Bar`
	TIP.PGE(RangeBase)
	TNT(0)
	TIP.PGD(RangeLimit+1)

IP Filtering and TraceStop

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.

35.2.5 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (PacketEn) is set. PacketEn is an internal state maintained in hardware in

response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

35.2.5.1 Packet Enable (PacketEn)

When PacketEn is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. PacketEn is composed of other states according to this relationship:

```
PacketEn ← TriggerEn AND ContextEn AND FilterEn AND BranchEn
```

These constituent controls are detailed in the following subsections.

PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and bookkeeping packets) may still be sent. See Section 35.2.6 for details of PacketEn and packet generation.

Note that, on processors that do not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0), FilterEn is treated as always set.

35.2.5.2 Trigger Enable (TriggerEn)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32_RTIT_CTL.TraceEn is set, and cleared by any of the following conditions:

- TraceEn is cleared by software.
- A TraceStop condition is encountered and IA32_RTIT_STATUS.Stopped is set.
- IA32_RTIT_STATUS.Error is set due to an operational error (see Section 35.3.9).

Software can discover the current TriggerEn value by reading the IA32_RTIT_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

35.2.5.3 Context Enable (ContextEn)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32_RTIT_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32_RTIT_STATUS.ContextEn bit. ContextEn is defined as follows:

```
ContextEn = !((IA32_RTIT_CTL.OS = 0 AND CPL = 0) OR
(IA32_RTIT_CTL.USER = 0 AND CPL > 0) OR (IS_IN_A_PRODUCTION_ENCLAVE[1]) OR
(IA32_RTIT_CTL.CR3Filter = 1 AND IA32_RTIT_CR3_MATCH does not match CR3)
```

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets (TNT, FUP, TIP.*, MODE.*) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as MTC and PSB (see Section 35.4.2.16 and Section 35.4.2.17), may still be generated while ContextEn is 0. For details of which packets are generated only when ContextEn is set, see Section 35.4.1.

The processor does not update ContextEn when TriggerEn = 0.

The value of ContextEn will toggle only when TriggerEn = 1.

35.2.5.4 Branch Enable (BranchEn)

This value is based purely on the IA32_RTIT_CTL.BranchEn value. If **BranchEn** is not set, then relevant COFI packets (TNT, TIP*, FUP, MODE.*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well

1. Trace packets generation is disabled in a production enclave, see Section 35.2.8.3. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

35.2.5.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32_RTIT_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 35.2.4.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 35.4.1.

After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32_RTIT_CTL.ADDRn_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

35.2.6 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32_RTIT_CTL (see Section 35.2.7.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-cache-able (UC) stores. Software should avoid using MTRRs to mark any portion of the Intel PT output region as UC, as this may override the behavior described above and force Intel PT stores to UC, thereby incurring severe performance impact.

There is no guarantee that a packet will be written to memory or other trace endpoint after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated have reached their endpoint is to clear TraceEn and follow that with a store, fence, or serializing instruction; doing so ensures that all buffered packets are flushed out of the processor.

35.2.6.1 Single Range Output

When IA32_RTIT_CTL.ToPA and IA32_RTIT_CTL.FabricEn bits are clear, trace packet output is sent to a single, contiguous memory (or MMIO if DRAM is not available) range defined by a base address in IA32_RTIT_OUTPUT_BASE (Section 35.2.7.7) and mask value in IA32_RTIT_OUTPUT_MASK_PTRS (Section 35.2.7.8). The current write pointer in this range is also stored in IA32_RTIT_OUTPUT_MASK_PTRS. This output range is circular, meaning that when the writes wrap around the end of the buffer they begin again at the base address.

This output method is best suited for cases where Intel PT output is either:

- Configured to be directed to a sufficiently large contiguous region of DRAM.

- Configured to go to an MMIO debug port, in order to route Intel PT output to a platform-specific trace endpoint (e.g., JTAG). In this scenario, a specific range of addresses is written in a circular manner, and SoC will intercept these writes and direct them to the proper device. Repeated writes to the same address do not overwrite each other, but are accumulated by the debugger, and hence no data is lost by the circular nature of the buffer.

The processor will determine the address to which to write the next trace packet output byte as follows:

```
OutputBase[63:0] ← IA32_RTIT_OUTPUT_BASE[63:0]
OutputMask[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[31:0])
OutputOffset[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[63:32])
trace_store_phys_addr ← (OutputBase & ~OutputMask) + (OutputOffset & OutputMask)
```

Single-Range Output Errors

If the output base and mask are not properly configured by software, an operational error (see Section 35.3.9) will be signaled, and tracing disabled. Error scenarios with single-range output are:

- Mask value is non-contiguous.

  IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTablePointer value has a 0 in a less significant bit position than the most significant bit containing a 1.
- Base address and Mask are mis-aligned, and have overlapping bits set.

  IA32_RTIT_OUTPUT_BASE && IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset > 0.
- Illegal Output Offset

  IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than the mask value (IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset).

Also note that errors can be signaled due to trace packet output overlapping with restricted memory, see Section 35.2.6.4.

35.2.6.2 Table of Physical Addresses (ToPA)

When IA32_RTIT_CTL.ToPA is set and IA32_RTIT_CTL.FabricEn is clear, the ToPA output mechanism is utilized. The ToPA mechanism uses a linked list of tables; see Figure 35-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- **proc_trace_table_base**.
  This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32_RTIT_OUTPUT_BASE MSR. While tracing is enabled, the processor updates the IA32_RTIT_OUTPUT_BASE MSR with changes to proc_trace_table_base, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_base.
- **proc_trace_table_offset**.
  This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads this value from bits 31:7 (MaskOrTableOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset with changes to proc_trace_table_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_offset.
- **proc_trace_output_offset**.
  This a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 (OutputOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates

IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset with changes to proc_trace_output_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_output_offset.

Figure 35-1 provides an illustration (not to scale) of the table and associated pointers.

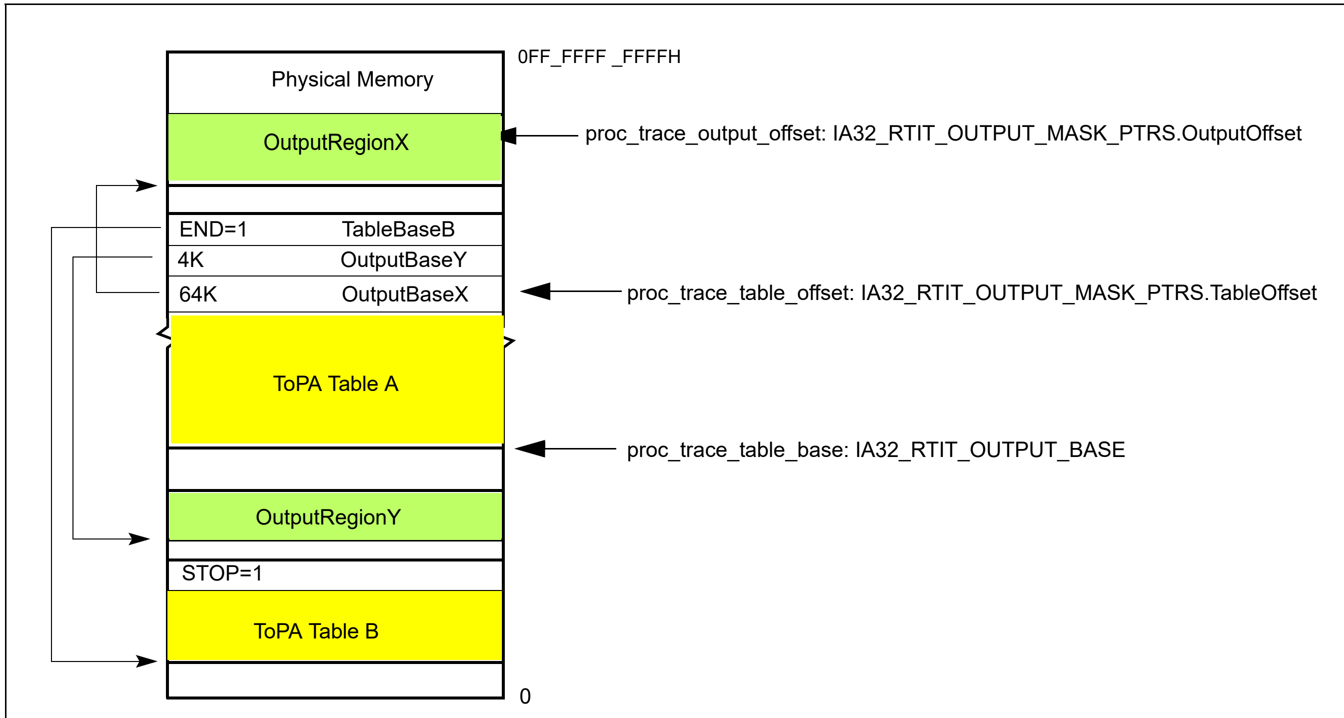


Figure 35-1. ToPA Memory Illustration

With the ToPA mechanism, the processor writes packets to the current output region (identified by proc_trace_table_base and the proc_trace_table_offset). The offset within that region to which the next byte will be written is identified by proc_trace_output_offset. When that region is filled with packet output (thus proc_trace_output_offset = RegionSize–1), proc_trace_table_offset is moved to the next ToPA entry, proc_trace_output_offset is set to 0, and packet writes begin filling the new output region specified by proc_trace_table_offset.

As packets are written out, each store derives its physical address as follows:

```
trace_store_phys_addr ← Base address from current ToPA table entry +
proc_trace_output_offset
```

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Section 35.2.6.2 for details on STOP.

When an END entry is reached, the processor loads proc_trace_table_base with the base address held in this END entry, thereby moving the current table pointer to this new table. The proc_trace_table_offset is reset to 0, as is the proc_trace_output_offset, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (proc_trace_table_offset = 01FFFFFH). In this case, the proc_trace_table_offset and proc_trace_output_offset are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs

while Intel PT is enabled may return stale values. Like all IA32_RTIT_* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32_RTIT_CTL.TraceEn. This ensures that he output MSR values account for all packets generated to that point, after which the output MSR values will be frozen until tracing resumes. [1]

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 35-2. The size of the address field is determined by the processor's physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

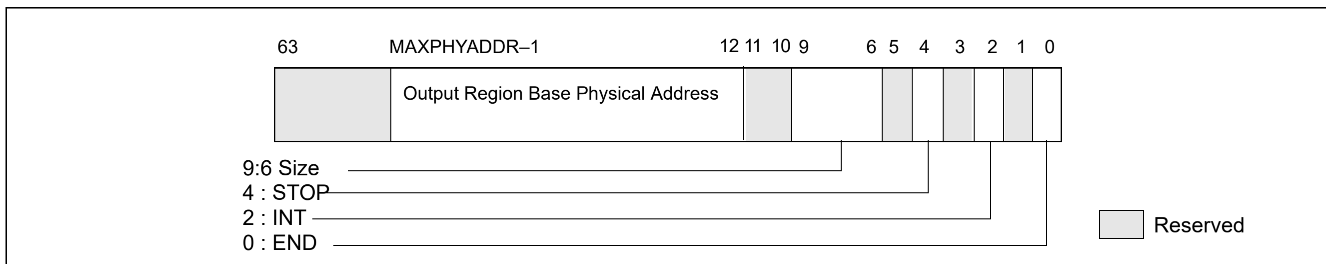


Figure 35-2. Layout of ToPA Table Entry

Table 35-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 35-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

Table 35-3. ToPA Table Entry Fields (Contd.)

ToPA Entry Field	Description
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.

ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32_RTIT_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 35.2.7.4 for details on IA32_RTIT_STATUS.Stopped. This sequence is known as "ToPA Stop".

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32_RTIT_OUTPUT_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset will hold the index value for that entry, and the IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset should be set to the size of the region.

Note that this means the offset pointer is pointing to the next byte after the end of the region, a configuration that would produce an operational error if the configuration remained when tracing is re-enabled with IA32_RTIT_STATUS.Stopped cleared.

ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 834H in x2APIC mode). See *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.
2. Set up an interrupt handler to service the interrupt vector that a ToPA PMI can raise.
3. Set the interrupt flag by executing STI.
4. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32_RTIT_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR (MSR 38EH). Once the ToPA PMI handler has serviced the relevant buffer, writing 1 to bit 55 of the MSR at 390H (IA32_GLOBAL_STATUS_RESET) clears IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI.

Intel PT is not frozen on PMI, and thus the interrupt handler will be traced (though filtering can prevent this). The Freeze_Perfmon_on_PMI and Freeze_LBRs_on_PMI settings in IA32_DEBUGCTL will be applied on ToPA PMI just as on other PMIs, and hence Perfmon counters are frozen.

Assuming the PMI handler wishes to read any buffered packets for persistent output, or wishes to modify any Intel PT MSRs, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

In rare cases, it may be possible to trigger a second ToPA PMI before the first is handled. This can happen if another ToPA region with INT=1 is filled before, or shortly after, the first PMI is taken, perhaps due to EFLAGS.IF being cleared for an extended period of time. This can manifest in two ways: either the second PMI is triggered before the first is taken, and hence only one PMI is taken, or the second is triggered after the first is taken, and thus will be taken when the handler for the first completes. Software can minimize the likelihood of the second case by clearing TraceEn at the beginning of the PMI handler. Further, it can detect such cases by then checking the Interrupt Request Register (IRR) for PMI pending, and checking the ToPA table base and off-set pointers (in IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) to see if multiple entries with INT=1 have been filled.

ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible that, in rare cases, the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 35-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32_RTIT_STATUS.Stopped indication before tracing can resume.

ToPA PMI and XSAVES/XRSTORS State Handling

In some cases the ToPA PMI may be taken after completion of an XSAVES instruction that switches Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, it is recommended that the Intel PT output configuration be modified by altering the ToPA tables themselves, rather than the Intel PT output MSRs.

Table 35-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

Table 35-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS

Pseudo Code Flow

```
IF (IA32_PERF_GLOBAL_STATUS.ToPA)
    Save IA32_RTIT_CTL value;
    IF ( IA32_RTIT_CTL.TraceEN )
        Disable Intel PT by clearing TraceEn;
    FI;
    IF ( there is space available to grow the current ToPA table )
        Add one or more ToPA entries after the last entry in the ToPA table;
        Point new ToPA entry address field(s) to new output region base(s);
    ELSE
        Modify an upcoming ToPA entry in the current table to have END=1;
        IF (output should transition to a new ToPA table )
            Point the address of the "END=1" entry of the current table to the new table base;
        ELSE
            /* Continue to use the current ToPA table, make a circular. */
            Point the address of the "END=1"l entry to the base of the current table;
            Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;
            /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */
        FI;
      FI;
      Restore saved IA32_RTIT_CTL.value;
FI;
```

ToPA Errors

When a malformed ToPA entry is found, an **operation error** results (see Section 35.3.9). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation.**
   This describes cases where a bit marked as reserved in Section 35.2.6.2 above is set to 1.
2. **ToPA alignment violation.**
   This includes cases where illegal ToPA entry base address bits are set to 1:
   a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32_RTIT_OUTPUT_BASE, or from a ToPA entry with END=1.
   b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
   c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset** (if IA32_RTIT_STATUS.Stopped=0).
   IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.
4. **ToPA rules violations.**
   These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
   a. Setting the STOP or INT bit on an entry with END=1.
   b. Setting the END bit in entry 0 of a ToPA table.
   c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
      i) ToPA table entry 1 with END=0.
      ii) ToPA table entry 1 with base address not matching the table base.

In all cases, the error will be logged by setting IA32_RTIT_STATUS.Error, thereby disabling tracing when the problematic ToPA entry is reached (when proc_trace_table_offset points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 35.2.6.4 for details.

A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 35.5.

35.2.6.3 Trace Transport Subsystem

When IA32_RTIT_CTL.FabricEn is set, the IA32_RTIT_CTL.ToPA bit is ignored, and trace output is written to the trace transport subsystem. The endpoints of this transport are platform-specific, and details of configuration options should refer to the specific platform documentation. The FabricEn bit is available to be set if CPUID(EAX=14H,ECX=0):EBX[bit 3] = 1.

35.2.6.4 Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 35-5 summarizes several scenarios.

Table 35-5. Behavior on Restricted Memory Access

Scenario	Description
ToPA output region overlaps with SMRR	Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 35.3.9) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).
ToPA table overlaps with SMRR	The processor will signal an error (Section 35.3.9) and disable tracing when the ToPA read pointer (IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset « 3)) enters the restricted region.

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the IA32_APIC_BASE MSR. For details about the APIC, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. No error is signaled for this case.

Modifications to Restricted Memory Regions

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing IA32_RTIT_CTL.TraceEn.

35.2.7 Enabling and Configuration MSRs

35.2.7.1 General Considerations

Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 35.3.1), RDMSR or WRMSR of the IA32_RTIT_* MSRs will cause #GP.
- A WRMSR to any of these configuration MSRs that begins and ends with IA32_RTIT_CTL.TraceEn set will #GP fault. Packet generation must be disabled before the configuration MSRs can be changed.

Note: Software may write the same value back to IA32_RTIT_CTL without #GP, even if TraceEn=1.

- All configuration MSRs for Intel PT are duplicated per logical processor
- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.
- All configuration MSRs for Intel PT are cleared on a cold RESET.
  - If CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 1, only the TraceEn bit is cleared on warm RESET; though this may have the impact of clearing other bits in IA32_RTIT_STATUS. Other MSR values of the trace configuration MSRs are preserved on warm RESET.
- The semantics of MSR writes to trace configuration MSRs in this chapter generally apply to explicit WRMSR to these registers, using VM-exit or VM-entry MSR load list to these MSRs, XRSTORS with requested feature bit map including XSAVE map component of state_8 (corresponding to IA32_XSS[bit 8]), and the write to IA32_RTIT_CTL.TraceEn by XSAVES (Section 35.3.5.2).

35.2.7.2 IA32_RTIT_CTL MSR

IA32_RTIT_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 35-6.

Table 35-6. IA32_RTIT_CTL MSR

Position	Bit Name	At Reset	Bit Description
0	TraceEn	0	If 1, enables tracing; else tracing is disabled if 0. When this bit transitions from 1 to 0, all buffered packets are flushed out of internal buffers. A further store, fence, or architecturally serializing instruction may be required to ensure that packet data can be observed at the trace endpoint. See Section 35.2.7.3 for details of enabling and disabling packet generation. Note that the processor will clear this bit on #SMI (Section) and warm reset. Other MSR bits of IA32_RTIT_CTL (and other trace configuration MSRs) are not impacted by these events.
1	CYCEn	0	0: Disables CYC Packet (see Section 35.4.2.14). 1: Enables CYC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
2	OS	0	0: Packet generation is disabled when CPL = 0. 1: Packet generation may be enabled when CPL = 0.
3	User	0	0: Packet generation is disabled when CPL > 0. 1: Packet generation may be enabled when CPL > 0.
4	PwrEvtEn	0	0: Power Event Trace packets are disabled. 1: Power Event Trace packets are enabled (see Section 35.2.3, “Power Event Tracing”).
5	FUPonPTW	0	0: PTW packets are not followed by FUPs. 1: PTW packets are followed by FUPs.
6	FabricEn	0	0: Trace output is directed to the memory subsystem, mechanism depends on IA32_RTIT_CTL.ToPA. 1: Trace output is directed to the trace transport subsystem, IA32_RTIT_CTL.ToPA is ignored. This bit is reserved if CPUID.(EAX=14H, ECX=0):ECX[bit 3] = 0.
7	CR3Filter	0	0: Disables CR3 filtering. 1: Enables CR3 filtering.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
8	ToPA	0	0: Single-range output scheme enabled if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 1 and IA32_RTIT_CTL.FabricEn=0. 1: ToPA output scheme enabled (see Section 35.2.6.2) if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 1, and IA32_RTIT_CTL.FabricEn=0. Note: WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit and FabricEn would cause #GP, if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 0. WRMSR to IA32_RTIT_CTL that sets this bit causes #GP, if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 0.
9	MTCEn	0	0: Disables MTC Packet (see Section 35.4.2.16). 1: Enables MTC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
10	TSCEn	0	0: Disable TSC packets. 1: Enable TSC packets (see Section 35.4.2.11).
11	DisRETC	0	0: Enable RET compression. 1: Disable RET compression (see Section 35.2.1.2).
12	PTWEn	0	0: PTWRITE packet generation disabled. 1: PTWRITE packet generation enabled (see Table 35-40 “PTW Packet Definition”).
13	BranchEn	0	0: Disable COFI-based packets. 1: Enable COFI-based packets: FUP, TIP, TIP.PGE, TIP.PGD, TNT, MODE.Exec, MODE.TSX. see Section 35.2.6 for details on BranchEn.
17:14	MTCFreq	0	Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined: 0: ART(0), 1: ART(1), 2: ART(2), 3: ART(3), 4: ART(4), 5: ART(5), 6: ART(6), 7: ART(7), 8: ART(8), 9: ART(9), 10: ART(10), 11: ART(11), 12: ART(12), 13: ART(13), 14: ART(14), 15: ART(15) Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
18	Reserved	0	Must be 0.
22:19	CycThresh	0	CYC packet threshold, see Section 35.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as $2^{(CycThresh-1)}$. The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384 Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
23	Reserved	0	Must be 0.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
27:24	PSBFreq	0	Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
31:28	Reserved	0	Must be 0.
35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] >= 0.
39:36	ADDR1_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR1_A/B based on the following encodings: 0: ADDR1 range unused. 1: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 2.
43:40	ADDR2_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 3.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
47:44	ADDR3_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 4.
59:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
63:60	Reserved	0	Must be 0.

35.2.7.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 35.4.2.17) will be generated if IA32_RTIT_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 35.4.2).

In addition to the packets discussed above, if and when PacketEn (Section 35.2.5.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a TIP.PGE packet (Section 35.4.2.3) will be generated.

When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the configuration of restricted memory regions). See Section 35.7 for more details of packets that may be generated with modifications to TraceEn.

Disabling Packet Generation

Clearing TraceEn causes any packet data buffered within the logical processor to be flushed out, after which the output MSRs (IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) will have stable values. When output is directed to memory, a store, fence, or architecturally serializing instruction may be required to ensure that the packet data is globally observed. No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

Other Writes to IA32_RTIT_CTL

Any attempt to modify IA32_RTIT_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32_RTIT_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

35.2.7.4 IA32_RTIT_STATUS MSR

The IA32_RTIT_STATUS MSR is readable and writable by software, but some bits (ContextEn, TriggerEn) are read-only and cannot be directly modified. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

Table 35-7. IA32_RTIT_STATUS MSR

Position	Bit Name	At Reset	Bit Description
0	FilterEn	0	This bit is written by the processor, and indicates that tracing is allowed for the current IP, see Section 35.2.5.5. Writes are ignored.
1	ContextEn	0	The processor sets this bit to indicate that tracing is allowed for the current context. See Section 35.2.5.3. Writes are ignored.
2	TriggerEn	0	The processor sets this bit to indicate that tracing is enabled. See Section 35.2.5.2. Writes are ignored.
3	Reserved	0	Must be 0.
4	Error	0	The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA Errors” in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
5	Stopped	0	The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA STOP” in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
31:6	Reserved	0	Must be 0.
48:32	PacketByteCnt	0	This field is written by the processor, and holds a count of packet bytes that have been sent out. The processor also uses this field to determine when the next PSB packet should be inserted. Note that the processor may clear or modify this field at any time while IA32_RTIT_CTL.TraceEn=1. It will have a stable value when IA32_RTIT_CTL.TraceEn=0. See Section 35.4.2.17 for details.
63:49	Reserved	0	Must be 0.

35.2.7.5 IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSRs

The role of the IA32_RTIT_ADDRn_A/B register pairs, for each n, is determined by the corresponding ADDRn_CFG fields in IA32_RTIT_CTL (see Section 35.2.7.2). The number of these register pairs is enumerated by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0].

- Processors that enumerate support for 1 range support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

- Processors that enumerate support for 2 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
- Processors that enumerate support for 3 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
- Processors that enumerate support for 4 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
  IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B

Each register has a single 64-bit field that holds a linear address value. Writes must ensure that the address is properly sign-extended, otherwise a #GP fault will result.

35.2.7.6 IA32_RTIT_CR3_MATCH MSR

The IA32_RTIT_CR3_MATCH register is compared against CR3 when IA32_RTIT_CTL.CR3Filter is 1. Bits 63:5 hold the CR3 address value to match, bits 4:0 are reserved to 0. For more details on CR3 filtering and the treatment of this register, see Section 35.2.4.2.

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32_RTIT_CR3_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

35.2.7.7 IA32_RTIT_OUTPUT_BASE MSR

This MSR is used to configure the trace output destination, when output is directed to memory (IA32_RTIT_CTL.FabricEn = 0). The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

When the ToPA output scheme is used, the processor may update this MSR when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-8. IA32_RTIT_OUTPUT_BASE MSR

Position	Bit Name	At Reset	Bit Description
6:0	Reserved	0	Must be 0.
MAXPHYADDR-1:7	BasePhysAddr	0	The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA: 0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value. The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 35.2.7.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 35.3.9). 1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See “ToPA Errors” in Section 35.2.6.2 as well as Section 35.3.9.
63:MAXPHYADDR	Reserved	0	Must be 0.

35.2.7.8 IA32_RTIT_OUTPUT_MASK_PTRS MSR

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 35.2.6.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. The size of this field indicates that regions can be of size 128B up to 4GB. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOrTableOffset field, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.

35.2.8 Interaction of Intel® Processor Trace and Other Processor Features

35.2.8.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 35-10 for details.

Table 35-10. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP)
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: ▪ Nested XBEGIN or XACQUIRE lock ▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set) ▪ Non-outermost XEND or XRELEASE lock	None. No change to TSX mode bits for these cases.

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

TSX and IP Filtering

A complication with tracking transactions is handling transactions that start or end outside of the tracing region. Transactions can't span across a change in ContextEn, because CPL changes and CR3 changes each cause aborts. But a transaction can start within the IP filter region and end outside it.

To assist the decoder handling this situation, MODE.TSX packets can be sent even if FilterEn=0, though there will be no FUP attached. Instead, they will merely serve to indicate to the decoder when transactions are active and when they are not. When tracing resumes (due to PacketEn=1), the last MODE.TSX preceding the TIP.PGE will indicate the current transaction status.

System Management Mode (SMM)

SMM code has special privileges that non-SMM code does not have. Intel Processor Trace can be used to trace SMM code, but special care is taken to ensure that SMM handler context is not exposed in any non-SMM trace collection. Additionally, packet output from tracing non-SMM code cannot be written into memory space that is either protected by SMRR or used by the SMM handler.

SMM is entered via a system management interrupt (SMI). SMI delivery saves the value of IA32_RTIT_CTL.TraceEn into SMRAM and then clears it, thereby disabling packet generation.

The saving and clearing of IA32_RTIT_CTL.TraceEn ensures two things:

1. All internally buffered packet data is flushed before entering SMM (see Section 35.2.7.2).
2. Packet generation ceases before entering SMM, so any tracing that was configured outside SMM does not continue into SMM. No SMM instruction pointers or other state will be exposed in the non-SMM trace.

When the RSM instruction is executed to return from SMM, the TraceEn value that was saved by SMI delivery is restored, allowing tracing to be resumed. As is done any time packet generation is enabled, ContextEn is re-evaluated, based on the values of CPL, CR3, etc., established by RSM.

Like other interrupts, delivery of an SMI produces a FUP containing the IP of the next instruction to execute. By toggling TraceEn, SMI and RSM can produce TIP.PGD and TIP.PGE packets, respectively, indicating that tracing was disabled or re-enabled. See Table 35.7 for more information about packets entering and leaving SMM.

Although #SMI and RSM change CR3, PIP packets are not generated in these cases. With #SMI tracing is disabled before the CR3 change; with RSM TraceEn is restored after CR3 is written.

TraceEn must be cleared before executing RSM, otherwise it will cause a shutdown. Further, on processors that restrict use of Intel PT with LBRs (see Section 35.3.1.2), any RSM that results in enabling of both will cause a shutdown.

Intel PT can support tracing of System Transfer Monitor operating in SMM, see Section 35.6.

35.2.8.2 Virtual-Machine Extensions (VMX)

Initial implementations of Intel Processor Trace do not support tracing in VMX operation. Such processors indicate this by returning 0 for IA32_VMX_MISC[bit 14]. On these processors, execution of the VMXON instruction clears IA32_RTIT_CTL.TraceEn and any attempt to set that bit in VMX operation using WRMSR causes a general-protection exception (#GP).

Processors that support Intel Processor Trace in VMX operation return 1 for IA32_VMX_MISC[bit 14]. Details of tracing in VMX operation are described in Section 35.5.

35.2.8.3 Intel Software Guard Extensions (SGX)

SGX provides an application with ability to instantiate a protective container (an enclave) with confidentiality and integrity (see *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. Enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.

During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 35.7.

Debug Enclaves

SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by enclave entry or exit. Hence the generation of ContextEn-dependent packets within a debug enclave is allowed.

35.2.8.4 SENTER/ENTERACCS and ACM

GETSEC[SENTER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

35.2.8.5 Intel® Memory Protection Extensions (Intel® MPX)

Bounds exceptions (#BR) caused by Intel MPX are treated like other exceptions, producing FUP and TIP packets that indicate the source and destination IPs.

35.3 CONFIGURATION AND PROGRAMMING GUIDELINE

35.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 35-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.

Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
EAX	31:0	Maximum valid sub-leaf Index	Specifies the index of the maximum valid sub-leaf for this CPUID leaf
EBX	0	CR3 Filtering Support	1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. See Section 35.2.7. 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CR3_MATCH, will #GP fault.
	1	Configurable PSB and Cycle-Accurate Mode Supported	1: (a) IA32_RTIT_CTL.PSBFreq can be set to a non-zero value, in order to select the preferred PSB frequency (see below for allowed values). (b) IA32_RTIT_STATUS.PacketByteCnt can be set to a non-zero value, and will be incremented by the processor when tracing to indicate progress towards the next PSB. If trace packet generation is enabled by setting TraceEn, a PSB will only be generated if PacketByteCnt=0. (c) IA32_RTIT_CTL.CYCEn can be set to 1 to enable Cycle-Accurate Mode. See Section 35.2.7. 0: (a) Any attempt to set IA32_RTIT_CTL.PSBFreq, to set IA32_RTIT_CTL.CYCEn, or write a non-zero value to IA32_RTIT_STATUS.PacketByteCnt any access to IA32_RTIT_CR3_MATCH, will #GP fault. (b) If trace packet generation is enabled by setting TraceEn, a PSB is always generated. (c) Any attempt to set IA32_RTIT_CTL.CYCEn will #GP fault.
	2	IP Filtering and TraceStop supported, and Preserve Intel PT MSRs across warm reset	1: (a) IA32_RTIT_CTL provides at one or more ADDRn_CFG field to configure the corresponding address range MSRs for IP Filtering or IP TraceStop. Each ADDRn_CFG field accepts a value in the range of 0:2 inclusive. The number of ADDRn_CFG fields is reported by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0]. (b) At least one register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B are provided to configure address ranges for IP filtering or IP TraceStop. (c) On warm reset, all Intel PT MSRs will retain their pre-reset values, though IA32_RTIT_CTL.TraceEn will be cleared. The Intel PT MSRs are listed in Section 35.2.7. 0: (a) An Attempt to write IA32_RTIT_CTL.ADDRn_CFG with non-zero encoding values will cause #GP. (b) Any access to IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B, will #GP fault. (c) On warm reset, all Intel PT MSRs will be cleared.
	3	MTC Supported	1: IA32_RTIT_CTL.MTCEn can be set to 1, and MTC packets will be generated. See Section 35.2.7. 0: An attempt to set IA32_RTIT_CTL.MTCEn or IA32_RTIT_CTL.MTCFreq to a non-zero value will #GP fault.
	4	PTWRITE Supported	1: Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. 0: Writes that set IA32_RTIT_CTL[12] or IA32_RTIT_CTL[5] will #GP, and PTWRITE will #UD fault.
	5	Power Event Trace Supported	1: Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. 0: Writes that set IA32_RTIT_CTL[4] will #GP.
	31:6	Reserved	

Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities (Contd.)

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
ECX	0	ToPA Output Supported	1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 35.2.6.2) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.SNGLRNGOUT[bit 2] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	1	ToPA Tables Allow Multiple Output Entries	1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 35.2.6.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see "ToPA Errors" in Section 35.2.6.2).
	2	Single-Range Output Supported	1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	3	Output to Trace Transport Subsystem Supported	1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will #GP fault.
	30:4	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.

Table 35-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=1)		Name	Description Behavior
Register	**Bits**		
EAX	2:0	Number of Address Ranges	A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. **NOTE:** Currently, no processors support more than 4 address ranges.
	15:3	Reserved	
	31:16	Bitmap of supported MTC Period Encodings	The non-zero bit positions indicate the map of supported encoding values for the IA32_RTIT_CTL.MTCFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 1 (MTC Packet generation is supported), otherwise the MTCFreq field is reserved to 0. Each bit position in this field represents 1 encoding value in the 4-bit MTCFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: MTCFreq can be assigned the associated encoding value. 0: MTCFreq cannot be assigned to the associated encoding value. A write to IA32_RTIT_CTLMTCFreq with unsupported encoding will cause #GP fault.
EBX	15:0	Bitmap of supported Cycle Threshold values	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.CycThresh field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Cycle-Accurate Mode is Supported), otherwise the CycThresh field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit CycThresh field (ie, bit 0 is associated with encoding value 0). For each bit: 1: CycThresh can be assigned the associated encoding value. 0: CycThresh cannot be assigned to the associated encoding value. A write to CycThresh with unsupported encoding will cause #GP fault.
	31:16	Bitmap of supported Configurable PSB Frequency encoding	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.PSBFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Configurable PSB is supported), otherwise the PSBFreq field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit PSBFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: PSBFreq can be assigned the associated encoding value. 0: PSBFreq cannot be assigned to the associated encoding value. A write to PSBFreq with unsupported encoding will cause #GP fault.
ECX	31:0	Reserved	
EDX	31:0	Reserved	

35.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an instruction pointer (IP) payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address (Note that in real mode, the CS base address is the value of CS<<4, while in protected mode the CS base address is the base linear address of the segment indicated by the CS register.). Which IP type is in use is indicated by enumeration (see CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31] in Table 35-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases where the CS base address is not 0 and the resolved LIP will not be evident in a trace generated on a CPU that enumerates use of RIP. This is likely to cause problems when attempting to link the trace with the associated binaries.

Note that IP comparison logic, for IP filtering and TraceStop range calculation, is based on the same IP type as these IP packets. For processors that output RIP, the IP comparison mechanism is also based on RIP, and hence on those processors RIP values should be written to IA32_RTIT_ADDRn_[AB] MSRs. This can produce differing behavior if the same trace configuration setting is run on processors reporting different IP types, i.e. CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31]. Care should be taken to check CPUID when configuring IP filters.

35.3.1.2 Model Specific Capability Restrictions

Some processor generations impose restrictions that prevent use of LBRs/BTS/BTM/LERs when software has enabled tracing with Intel Processor Trace. On these processors, when TraceEn is set, updates of LBR, BTS, BTM, LERs are suspended but the states of the corresponding IA32_DEBUGCTL control fields remained unchanged as if it were still enabled. When TraceEn is cleared, the LBR array is reset, and LBR/BTS/BTM/LERs updates will resume. Further, reads of these registers will return 0, and writes will be dropped.

The list of MSRs whose updates/accesses are restricted follows.

- MSR_LASTBRANCH_x_TO_IP, MSR_LASTBRANCH_x_FROM_IP, MSR_LBR_INFO_x, MSR_LASTBRANCH_TOS
- MSR_LER_FROM_IP, MSR_LER_TO_IP
- MSR_LBR_SELECT

For processor with CPUID DisplayFamily_DisplayModel signature of 06_3DH, 06_47H, 06_4EH, 06_4FH, 06_56H and 06_5EH, the use of Intel PT and LBRs are mutually exclusive.

35.3.2 Enabling and Configuration of Trace Packet Generation

To configure trace packets, enable packet generation, and capture packets, software starts with using CPUID instruction to detect its feature flag, CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1; followed by enumerating the capabilities described in Section 35.3.1.

Based on the capability queried from Section 35.3.1, software must configure a number of model-specific registers. This section describes programming considerations related to those MSRs.

35.3.2.1 Enabling Packet Generation

When configuring and enabling packet generation, the IA32_RTIT_CTL MSR should be written after any other Intel PT MSRs have been written, since writes to the other configuration MSRs cause a general-protection fault (#GP) if TraceEn = 1. If a prior trace collection context is not being restored, then software should first clear IA32_RTIT_STATUS. This is important since the Stopped, and Error fields are writable; clearing the MSR clears any values that may have persisted from prior trace packet collection contexts. See Section 35.2.7.2 for details of packets generated by setting TraceEn to 1.

If setting TraceEn to 1 causes an operational error (see Section 35.3.9), there may be a delay after the WRMSR completes before the error is signaled in the IA32_RTIT_STATUS MSR.

While packet generation is enabled, the values of some configuration MSRs (e.g., IA32_RTIT_STATUS and IA32_RTIT_OUTPUT_*) are transient, and reads may return values that are out of date. Only after packet generation is disabled (by clearing TraceEn) do reads of these MSRs return reliable values.

35.3.2.2 Disabling Packet Generation

After disabling packet generation by clearing IA32_RTIT_CTL, it is advisable to read the IA32_RTIT_STATUS MSR (Section 35.2.7.4):

- If the Error bit is set, an operational error was encountered, and the trace is most likely compromised. Software should check the source of the error (by examining the output MSR values), correct the source of the problem, and then attempt to gather the trace again. For details on operational errors, see Section 35.3.9. Software should clear IA32_RTIT_STATUS.Error before re-enabling packet generation.
- If the Stopped bit is set, software execution encountered an IP TraceStop (see Section 35.2.4.3) or the ToPA Stop condition (see "ToPA STOP" in Section 35.2.6.2) before packet generation was disabled.

35.3.3 Flushing Trace Output

Packets are first buffered internally and then written out asynchronously. To collect packet output for post-processing, a collector needs first to ensure that all packet data has been flushed from internal buffers. Software can ensure this by stopping packet generation by clearing IA32_RTIT_CTL.TraceEn (see "Disabling Packet Generation" in Section 35.2.7.2).

When software clears IA32_RTIT_CTL.TraceEn to flush out internally buffered packets, the logical processor issues an SFENCE operation which ensures that WC trace output stores will be ordered with respect to the next store, or serializing operation. A subsequent read from the same logical processor will see the flushed trace data, while a read from another logical processor should be preceded by a store, fence, or architecturally serializing operation on the tracing logical processor.

When the flush operations complete, the IA32_RTIT_OUTPUT_* MSR values indicate where the trace ended. While TraceEn is set, these MSRs may hold stale values. Further, if a ToPA region with INT=1 is filled, meaning a ToPA PMI has been triggered, IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI[55] will be set by the time the flush completes.

35.3.4 Warm Reset

The MSRs software uses to program Intel Processor Trace are cleared after a power-on RESET (or cold RESET). On a warm RESET, the contents of those MSRs can retain their values from before the warm RESET with the exception that IA32_RTIT_CTL.TraceEn will be cleared (which may have the side effect of clearing some bits in IA32_RTIT_STATUS).

35.3.5 Context Switch Consideration

To facilitate construction of instruction execution traces at the granularity of a software process or thread context, software can save and restore the states of the trace configuration MSRs across the process or thread context switch boundary. The principle is the same as saving and restoring the typical architectural processor states across context switches.

35.3.5.1 Manual Trace Configuration Context Switch

The configuration can be saved and restored through a sequence of instructions of RDMSR, management of MSR content and WRMSR. To stop tracing and to ensure that all configuration MSRs contain stable values, software must clear IA32_RTIT_CTL.TraceEn before reading any other trace configuration MSRs. The recommended method for saving trace configuration context manually follows:

1. RDMSR IA32_RTIT_CTL, save value to memory
2. WRMSR IA32_RTIT_CTL with saved value from RDMSR above and TraceEn cleared
3. RDMSR all other configuration MSRs whose values had changed from previous saved value, save changed values to memory

When restoring the trace configuration context, IA32_RTIT_CTL should be restored last:

1. Read saved configuration MSR values, aside from IA32_RTIT_CTL, from memory, and restore them with WRMSR
2. Read saved IA32_RTIT_CTL value from memory, and restore with WRMSR.

35.3.5.2 Trace Configuration Context Switch Using XSAVES/XRSTORS

On processors whose XSAVE feature set supports XSAVES and XRSTORS, the Trace configuration state can be saved using XSAVES and restored by XRSTORS, in conjunction with the bit field associated with supervisory state component in IA32_XSS. See Chapter 13, "Managing State Using the XSAVE Feature Set" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The layout of the trace configuration component state in the XSAVE area is shown in Table 35-13.[1]

Table 35-13. Memory Layout of the Trace Configuration State Component

Offset within Component Area	Field	Offset within Component Area	Field
0H	IA32_RTIT_CTL	08H	IA32_RTIT_OUTPUT_BASE
10H	IA32_RTIT_OUTPUT_MASK_PTRS	18H	IA32_RTIT_STATUS
20H	IA32_RTIT_CR3_MATCH	28H	IA32_RTIT_ADDR0_A
30H	IA32_RTIT_ADDR0_B	38H	IA32_RTIT_ADDR1_A
40H	IA32_RTIT_ADDR1_B	48H–End	Reserved

The IA32_XSS MSR is zero coming out of RESET. Once IA32_XSS[bit 8] is set, system software operating at CPL= 0 can use XSAVES/XRSTORS with the appropriate requested-feature bitmap (RFBM) to manage supervisor state components in the XSAVE map. See Chapter 13, "Managing State Using the XSAVE Feature Set" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

35.3.6 Cycle-Accurate Mode

Intel PT can be run in a cycle-accurate mode which enables CYC packets (see Section 35.4.2.14) that provide low-level information in the processor core clock domain. This cycle counter data in CYC packets can be used to compute IPC (Instructions Per Cycle), or to track wall-clock time on a fine-grain level.

To enable cycle-accurate mode packet generation, software should set IA32_RTIT_CTL.CYCEn=1. It is recommended that software also set TSCEn=1 anytime cycle-accurate mode is in use. With this, all CYC-eligible packets will be preceded by a CYC packet, the payload of which indicates the number of core clock cycles since the last CYC packet. In cases where multiple CYC-eligible packets are generated in a single cycle, only a single CYC will be generated before the CYC-eligible packets, otherwise each CYC-eligible packet will be preceded by its own CYC. The CYC-eligible packets are:

- TNT, TIP, TIP.PGE, TIP.PGD, MODE.EXEC, MODE.TSX, PIP, VMCS, OVF, MTC, TSC, PTWRITE, EXSTOP

TSC packets are generated when there is insufficient information to reconstruct wall-clock time, due to tracing being disabled (TriggerEn=0), or power down scenarios like a transition to a deep-sleep MWAIT C-state. In this case, the CYC that is generated along with the TSC will indicate the number of cycles actively tracing (those powered up, with TriggerEn=1) executed between the last CYC packet and the TSC packet. And hence the amount of time spent while tracing is inactive can be inferred from the difference in time between that expected based on the CYC value, and the actual time indicated by the TSC.

Additional CYC packets may be sent stand-alone, so that the processor can ensure that the decoder is aware of the number of cycles that have passed before the internal hardware counter wraps, or is reset due to other micro-architectural condition. There is no guarantee at what intervals these standalone CYC packets will be sent, except that they will be sent before the wrap occurs. An illustration is given below.

1. Table 35-13 documents support for the MSRs defining address ranges 0 and 1. Processors that provide XSAVE support for Intel Processor Trace support only those address ranges.

Example 35-1. An Illustrative CYC Packet Example

Time (cycles)	Instruction Snapshot	Generated Packets	Comment
x	call %eax	CYC(?), TIP	?Elapsed cycles from the previous CYC unknown
x + 2	call %ebx	CYC(2), TIP	1 byte CYC packet; 2 cycles elapsed from the previous CYC
x + 8	jnz Foo (not taken)	CYC(6)	1 byte CYC packet
x + 9	ret (compressed)		
x + 12	jnz Bar (taken)		
x + 16	ret (uncompressed)	TNT, CYC(8), TIP	1 byte CYC packet
x + 4111		CYC(4095)	2 byte CYC packet
x + 12305		CYC(8194)	3 byte CYC packet
x + 16332	mov cr3, %ebx	CYC(4027), PIP	2 byte CYC packet

35.3.6.1 Cycle Counter

The cycle counter is implemented in hardware (independent of the time stamp counter or performance monitoring counters), and is a simple incrementing counter that does not saturate, but rather wraps. The size of the counter is implementation specific.

The cycle counter is reset to zero any time that TriggerEn is cleared, and when a CYC packet is sent. The cycle counter will continue to count when ContextEn or FilterEn are cleared, and cycle packets will still be generated. It will not count during sleep states that result in Intel PT logic being powered-down, but will count up to the point where clocks are disabled, and resume counting once they are re-enabled.

35.3.6.2 Cycle Packet Semantics

Cycle-accurate mode adheres to the following protocol:

- All packets that precede a CYC packet represent instructions or events that took place before the CYC time.
- All packets that follow a CYC packet represent instructions or events that took place at the same time as, or after, the CYC time.
- The CYC-eligible packet that immediately follows a CYC packet represents an instruction or event that took place at the same time as the CYC time.

These items above give the decoder a means to apply CYC packets to a specific instruction in the assembly stream. Most packets represent a single instruction or event, and hence the CYC packet that precedes each of those packets represents the retirement time of that instruction or event. In the case of TNT packets, up to 6 conditional branches and/or compressed RETs may be contained in the packet. In this case, the preceding CYC packet provides the retirement time of the first branch in the packet. It is possible that multiple branches retired in the same cycle as that first branch in the TNT, but the protocol will not make that obvious. Also note that a MTC packet could be generated in the same cycle as the first JCC in the TNT packet. In this case, the CYC would precede both the MTC and the TNT, and apply to both.

Note that there are times when the cycle counter will stop counting, though cycle-accurate mode is enabled. After any such scenario, a CYC packet followed by TSC packet will be sent. See Section 35.8.3.2 to understand how to interpret the payload values

Multi-packet Instructions or Events

Some operations, such as interrupts or task switches, generate multiple packets. In these cases, multiple CYC packets may be sent for the operation, preceding each CYC-eligible packet in the operation. An example, using a task switch on a software interrupt, is shown below.

Example 35-2. An Example of CYC in the Presence of Multi-Packet Operations

Time (cycles)	Instruction Snapshot	Generated Packets
x	jnz Foo (not taken)	CYC(?),
x + 2	ret (compressed)	
x + 8	jnz Bar (taken)	
x + 9	jmp %eax	TNT, CYC(9), TIP
x + 12	jnz Bar (not taken)	CYC(3)
x + 32	int3 (task gate)	TNT, FUP, CYC(10), PIP, CYC(20), MODE.Exec, TIP

35.3.6.3 Cycle Thresholds

Software can opt to reduce the frequency of cycle packets, a trade-off to save bandwidth and intrusion at the expense of precision. This is done by utilizing a cycle threshold (see Section 35.2.7.2).

IA32_RTIT_CTL.CycThresh indicates to the processor the minimum number of cycles that must pass before the next CYC packet should be sent. If this value is 0, no threshold is used, and CYC packets can be sent every cycle in which a CYC-eligible packet is generated. If this value is greater than 0, the hardware will wait until the associated number of cycles have passed since the last CYC packet before sending another. CPUID provides the threshold options for CycThresh, see Section 35.3.1.

Note that the cycle threshold does not dictate how frequently a CYC packet will be posted, it merely assigns the maximum frequency. If the cycle threshold is 16, a CYC packet can be posted no more frequently than every 16 cycles. However, once that threshold of 16 cycles has passed, it still requires a new CYC-eligible packet to be generated before a CYC will be inserted. Table 35-14 illustrates the threshold behavior.

Table 35-14. An Illustrative CYC Packet Example

Time (cycles)	Instruction Snapshot	Threshold			
		0	16	32	64
x	jmp %eax	CYC, TIP	CYC, TIP	CYC, TIP	CYC, TIP
x + 9	call %ebx	CYC, TIP	TIP	TIP	TIP
x + 15	call %ecx	CYC, TIP	TIP	TIP	TIP
x + 30	jmp %edx	CYC, TIP	CYC, TIP	TIP	TIP
x + 38	mov cr3, %eax	CYC, PIP	PIP	CYC, PIP	PIP
x + 46	jmp [%eax]	CYC, TIP	CYC, TIP	TIP	TIP
x + 64	call %edx	CYC, TIP	CYC, TIP	TIP	CYC,TIP
x + 71	jmp %edx	CYC, TIP	TIP	CYC,TIP	TIP

35.3.7 Decoder Synchronization (PSB+)

The PSB packet (Section 35.4.2.17) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well. The decoder should never need to retain any information (e.g., LastIP, call stack, compound packet event) across a PSB; all compound packet events will be completed before a PSB, and any compression state will be reset.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 35.4.2.18). One or more packets may be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as “status only”, since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the

normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:

- Timestamp (TSC), if IA32_RTIT_CTL.TSCEn=1.
- Timestamp-MTC Align (TMA), if IA32_RTIT_CTL.TSCEn=1 && IA32_RTIT_CTL.MTCEn=1.
- Paging Info Packet (PIP), if ContextEn=1 and IA32_RTIT_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the "conceal VMX non-root operation from Intel PT". VM-execution control is 0.
- VMCS packet, if either the logical is in VMX root operation or the logical processor is in VMX non-root operation and the "conceal VMX non-root operation from Intel PT" VM-execution control is 0.
- Core Bus Ratio (CBR).
- MODE.TSX, if ContextEn=1 and BranchEn = 1.
- MODE.Exec, if PacketEn=1.
- Flow Update Packet (FUP), if PacketEn=1.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies. The ordering of packets within PSB+ is not fixed. Timing packets such as CYC and MTC may be generated between PSB and PSBEND, and their meanings are the same as outside PSB+.

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+.

35.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor's dedicated internal buffers are all full, an "internal buffer overflow" occurs. On such an overflow packet generation ceases (as packets would need to enter the processor's internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 35.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32_RTIT_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. Between the OVF and following FUP or TIP.PGE, there may be packets that do not depend on PacketEn, such as timing packets. If the overflow resolves while PacketEn=0, other packets that are not dependent on PacketEn may come before the TIP.PGE.

35.3.8.1 Overflow Impact on Enables

The address comparisons to ADDRn ranges, for IP filtering and TraceStop (Section 35.2.4.3), continue during a buffer overflow, and TriggerEn, ContextEn, and FilterEn may change during a buffer overflow. Like other packets, however, any TIP.PGE or TIP.PGD packets that would have been generated will be lost. Further, IA32_RTIT_STATUS.PacketByteCnt will not increment, since it is only incremented when packets are generated.

If a TraceStop event occurs during the buffer overflow, IA32_RTIT_STATUS.Stopped will still be set, tracing will cease as a result. However, the TraceStop packet, and any TIP.PGD that result from the TraceStop, may be dropped.

35.3.8.2 Overflow Impact on Timing Packets

Any timing packets that are generated during a buffer overflow will be dropped. If only a few MTC packets are dropped, a decoder should be able to detect this by noticing that the time value in the first MTC packet after the buffer overflow incremented by more than one. If the buffer overflow lasted long enough that 256 MTC packets are lost (and thus the MTC packet 'wraps' its 8-bit CTC value), then the decoder may be unable to properly understand

the trace. This is not an expected scenario. No CYC packets are generated during overflow, even if the cycle counter wraps.

Note that, if cycle-accurate mode is enabled, the OVF packet will generate a CYC packet. Because the cycle counter counts during overflows, this CYC packet can provide the duration of the overflow. However, there is a risk that the cycle counter wrapped during the overflow, which could render this CYC misleading.

35.3.9 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See "ToPA Errors" in Section 35.2.6.2 for details on ToPA errors, and Section 35.2.6.4 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32_RTIT_STATUS.Error is set, which will cause IA32_RTIT_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32_RTIT_STATUS.Error. At this point, packet generation can be re-enabled.

35.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

35.4.1 Packet Relationships and Ordering

This section introduces the concept of packet "binding", which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 35.4.2 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of "compound packet events", such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be "coupled" with the TIP packet.

Other packets could be in between the coupled FUP and TIP packet. Timing packets, such as TSC, MTC, CYC, or CBR, could arrive at any time, and hence could intercede in a compound packet event. If an operation changes CR3 or the processor's mode of execution, a state update packet (i.e., PIP or MODE) is generated. The state changes indicated by these intermediate packets should be applied at the IP of the TIP* packet. A summary of compound packet events is provided in Table 35-15; see Section 35.4.2 for more per-packet details and Section 35.7 for more detailed packet generation examples.

Table 35-15. Compound Packet Event Summary

Event Type	Beginning	Middle	End	Comment
Unconditional, uncompressed control-flow transfer	FUP or none	Any combination of PIP, VMCS, MODE.Exec, or none	TIP or TIP.PGD	FUP only for asynchronous events. Order of middle packets may vary. PIP/VMCS/MODE only if the operation modifies the state tracked by these respective packets
TSX Update	MODE.TSX, and (FUP or none)	None	TIP, TIP.PGD, or none	FUP TIP/TIP.PGD only for TSX abort cases
Overflow	OVF	PSB, PSBEND, or none	FUP or TIP.PGE	FUP if overflow resolves while ContextEn=1, else TIP.PGE.

35.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 35-3 explains how to interpret them. Packet bits listed as "RSVD" are not guaranteed to be 0.

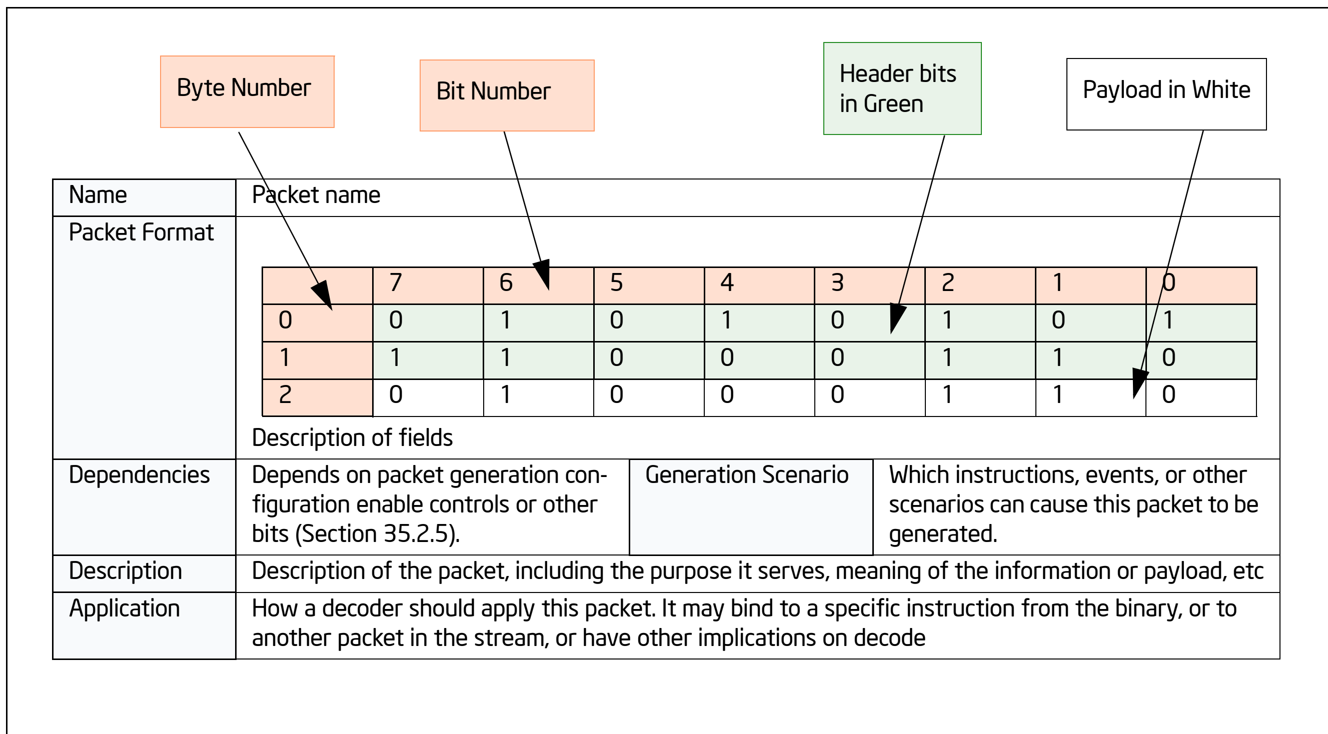


Figure 35-3. Interpreting Tabular Definition of Packet Format

35.4.2.1 Taken/Not-taken (TNT) Packet

Table 35-16. TNT Packet Definition

Name	Taken/Not-taken (TNT) Packet
Packet Format	

	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

Table 35-16. TNT Packet Definition (Contd.)

B1...BN represent the last N conditional branch or compressed RET (Section 35.4.2.2) results, such that B1 is oldest and BN is youngest. The short TNT packet can contain from 1 to 6 TNT bits. The long TNT packet can contain up from 1 to 47 TNT bits.

	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	Long TNT
1	1	0	1	0	0	0	1	1	
2	B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}	
3	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}	B_{38}	B_{39}	
4	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}	
5	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	
6	B_{8}	B_{9}	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	
7	1	B_{1}	B_{2}	B_{3}	B_{4}	B_{5}	B_{6}	B_{7}	

Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these “partial TNTs” are shown below.

	7	6	5	4	3	2	1	0	
0	0	0	1	B_{1}	B_{2}	B_{3}	B_{4}	0	Short TNT

	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	Long TNT
1	1	0	1	0	0	0	1	1	
2	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}	
3	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	
4	B_{8}	B_{9}	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	
5	1	B_{1}	B_{2}	B_{3}	B_{4}	B_{5}	B_{6}	B_{7}	
6	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	

Dependencies	PacketEn	Generation Scenario	On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated, or certain micro-architectural conditions occurring, before the TNT is full.
Description	Provides the taken/not-taken results for the last 1–N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 35.4.2.2). The TNT payload bits should be interpreted as follows: ▪ 1 indicates a taken conditional branch, or a compressed RET ▪ 0 indicates a not-taken conditional branch		
Application	Each valid payload bit (that is, bits between the header bits and the trailing Stop bit) applies to an upcoming conditional branch or RET instruction. Once a decoder consumes a TNT packet with N valid payload bits, these bits should be applied to (and hence provide the destination for) the next N conditional branches or RETs		

35.4.2.2 Target IP (TIP) Packet

Table 35-17. IP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Target IP (TIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td>Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, (VM exit, VM entry,)[1] TSX abort, (EENTER, EEXIT, ERESUME, AEX)[2].</td></tr>
<tr><td>Description</td><td colspan="3">Provides the target for some control flow transfers</td></tr>
<tr><td rowspan="2">Application</td><td colspan="3">Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</td></tr>
<tr><td colspan="3">The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.
2. In a debug enclave.

IP Compression

The IP payload in a TIP. FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the "Last IP" that was encoded in trace packets, thus the decoder will need to keep track of the "Last IP" state in software, to match fidelity with packets generated by hardware. "Last IP" is initialized to zero, hence if the first IP in the trace may be compressed if the upper bytes are zeroes.

The "IPBytes" field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

Table 35-18. FUP/TIP IP Reconstruction

IPBytes	Uncompressed IP Value							
	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
000b	None, IP is out of context							
001b	Last IP[63:16]						IP Payload[15:0]	
010b	Last IP[63:32]				IP Payload[31:0]			
011b	IP Payload[47] extended		IP Payload[47:0]					
100b	Last IP [63:48]		IP Payload[47:0]					
101b	Reserved							
110b	IP Payload[63:0]							
111b	Reserved							

The processor-internal Last IP state is guaranteed to be reset to zero when a PSB is sent out. This means that the IP that follows the PSB with either be un-compressed (011b or 110b, see Table 35-18), or compressed against zero.

At times, “IPbytes” will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code. In that case, no TargetIP will be included in the packet, since that would expose an instruction point at CPL = 0. When the IP payload is suppressed in this manner, Last IP is not cleared, and instead refers to the last IP packet with a non-zero IPBytes field.

On processors that support a maximum linear address size of 32 bits, IP payloads may never exceed 32 bits (IPBytes <= 010b).

Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don’t have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the target of the RET.

In cases where the RET is compressed, the target is guaranteed to match the value produced in 2) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from 2).

The hardware ensure that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because it sends no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32_RTIT_CTL.DisRETC). For processors that employ deferred TIPs (Section 35.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make

clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL's Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior

35.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

Table 35-19. TNT Examples with Deferred TIPs

Code Flow	Packets, Non-Deferred TIPS	Packets, Deferred TIPS
0x1000 cmp %rcx, 0 0x1004 jnz Foo // not-taken 0x1008 jmp %rdx	TNT(0b0), TIP(0x1308)	
0x1308 cmp %rcx, 1 0x130c jnz Bar // not-taken 0x1310 cmp %rcx, 2 0x1314 jnz Baz // taken 0x1500 cmp %eax, 7 0x1504 jg Exit // not-taken 0x1508 jmp %r15	TNT(0b010), TIP(0x1100)	
0x1100 cmp %rbx, 1 0x1104 jg Start // not-taken 0x1108 add %rcx, %eax 0x110c ... // **an asynchronous Interrupt arrives** INThandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)

35.4.2.4 Packet Generation Enable (TIP.PGE)

Table 35-20. TIP.PGE Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Target IP - Packet Generation Enable (TIP.PGE)</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 1</td><td>Generation Scenario</td><td>Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn</td></tr>
<tr><td>Description</td><td colspan="3">Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins.
This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:
▪ TriggerEn: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR.
▪ FilterEn: This is set when software jumps into the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will be the target of the branch.
▪ ContextEn: This is set on a CPL change, a CR3 write or any other means of changing ContextEn. The IP payload will be the Next IP of the instruction that changes context if it is not a branch, otherwise it will be the target of the branch.</td></tr>
<tr><td>Application</td><td colspan="3">TIP.PGE packets bind to the instruction at the IP given in the payload.</td></tr>
</table>

35.4.2.5 Packet Generation Disable (TIP.PGD)

Table 35-21. TIP.PGD Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Target IP - Packet Generation Disable (TIP.PGD)</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 0</td><td>Generation Scenario</td><td>Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn</td></tr>
<tr><td>Description</td><td colspan="3">Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn= 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn.
PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:
• TriggerEn: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or when IA32_RTIT_STATUS.Stopped is set, or on operational error. The IP payload will be suppressed in this case, and the "IPBytes" field will have the value 0.
• FilterEn: This is set when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes = 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.
• ContextEn: This can happen on a CPL change, a CR3 write or any other means of changing ContextEn. See Section 35.2.4.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The "IPBytes" field will have value 0.
Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNI bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (ie, SYSCALL when IA32_RTIT_CTL.OS=0, In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.</td></tr>
<tr><td>Application</td><td colspan="3">TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce.
In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.
If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.</td></tr>
</table>

35.4.2.6 Flow Update (FUP) Packet

Table 35-22. FUP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Flow Update (FUP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">IP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">IP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">IP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">IP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">IP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">IP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">IP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">IP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 35.2.4 for details.)</td><td>Generation Scenario</td><td>Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit[1], #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, (EENTRY, EEXIT, ERESUME, EEE, AEX,)[2], INT 0, INT 3, INT n, a WRMSR that disables packet generation.</td></tr>
<tr><td>Description</td><td colspan="3">Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.</td></tr>
<tr><td>Application</td><td colspan="3">FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.
In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 35.4.2.8 for details.
Otherwise, FUPs are part of compound packet events (see Section 35.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.
2. If Intel Software Guard Extensions is supported.

FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 35-23 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

Table 35-23. FUP Cases and IP Payload

Event	Flow Update IP	Comment
External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), INIT/SIPI	Address of next instruction (Next IP) that would have been executed	Functionally, this matches the LBR FROM field value and also the EIP value which is saved onto the stack.
Exceptions/Faults, Machine check (fault-like)	Address of the instruction which took the exception/fault (Current IP)	This matches the similar functionality of LBR FROM field value and also the EIP value which is saved onto the stack.
Software Interrupt	Address of the software interrupt instruction (Current IP)	This matches the similar functionality of LBR FROM field value, but does not match the EIP value which is saved onto the stack (Next Linear Instruction Pointer - NLIP).
EENTER, EEXIT, ERESUME, Enclave Exiting Event (EEE), AEX[1]	Current IP of the instruction	This matches the LBR FROM field value and also the EIP value which is saved onto the stack.
XACQUIRE	Address of the X* instruction	
XRELEASE, XBEGIN, XEND, XABORT, other transactional abort	Current IP	
#SMI	IP that is saved into SMRAM	
WRMSR that clears TraceEn	Current IP	

NOTES:

1. Information on EENTER, EEXIT, ERESUME, EEE, Asynchronous Enclave eXit (AEX) can be found in *Intel® Software Guard Extensions Programming Reference*.

On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

If there are post-commit task switch faults, the IP value of the FUP will be the original IP when the task switch started. This is the same value as would be seen in the LBR_FROM field. But it is a different value as is saved on the stack or VMCS.

35.4.2.7 Paging Information (PIP) Packet

Table 35-24. PIP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Paging Information (PIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="7">CR3[11:5] or 0</td><td>RSVD/NR</td></tr>
<tr><td>3</td><td colspan="8">CR3[19:12]</td></tr>
<tr><td>4</td><td colspan="8">CR3[27:20]</td></tr>
<tr><td>5</td><td colspan="8">CR3[35:28]</td></tr>
<tr><td>6</td><td colspan="8">CR3[43:36]</td></tr>
<tr><td>7</td><td colspan="8">CR3[51:44]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn &&
IA32_RTIT_CTL.OS</td><td>Generation
Scenario</td><td>MOV CR3, Task switch, INIT, SIPI, PSB+;
If IA32_VMX_MISC[bit 14] reports 1: VM exit, VM entry</td></tr>
<tr><td>Description</td><td colspan="3">The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other page modes (32-bit and 4-level paging[1]), these bits are 0.
This packet holds the CR3 address value. It will be generated on operations that modify CR3:
▪ MOV CR3 operation
▪ Task Switch
▪ INIT and SIPI
▪ VM exit and VM entry, if appropriate controls in the VMCS are clear (see Section 35.5.1)
PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.
The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.
The PIP packet contains the new CR3 value when CR3 is written.
PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX non-root operation from Intel PT” VM-execution control is 0. All other PIPs clear the NR bit.</td></tr>
<tr><td>Application</td><td colspan="3">The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time.
When a PIP is encountered, a decoder should do the following:
1) If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new CR3/NR values to the TIP payload IP.
2) Otherwise, look for the next MOV CR3, far branch, or VMRESUME/VMLAUNCH in the disassembly, and apply the new CR3 to the next (or target) IP.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

NOTES:

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

35.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

Table 35-25. General Form of MODE Packets

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	Leaf ID			Mode				

The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

MODE.Exec Packet

Table 35-26. MODE.Exec Packet Definition

<table>
<tr><td>Name</td><td colspan="2">MODE.Exec Packet</td></tr>
<tr><td>Packet Format</td><td colspan="2"><table><tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>CS.D</td><td>(CS.L & LMA)</td></tr></table></td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario: Far branch, interrupt, exception, (VM exit, VM entry,)[1] if the mode changes.
PSB+, and any scenario that can generate a TIP.PGE, such that the mode may have changed since the last MODE.Exec.</td></tr>
<tr><td>Description</td><td colspan="2">Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L & IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.
<table><tr><td>CS.D</td><td>(CS.L & IA32_EFER.LMA)</td><td>Addressing Mode</td></tr><tr><td>1</td><td>1</td><td>N/A</td></tr><tr><td>0</td><td>1</td><td>64-bit mode</td></tr><tr><td>1</td><td>0</td><td>32-bit mode</td></tr><tr><td>0</td><td>0</td><td>16-bit mode</td></tr></table>
MODE.Exec is sent at the time of a mode change, if PacketEn=1 at the time, or when tracing resumes, if necessary. In the former case, the MODE.Exec packet is generated along with other packets that result from the far transfer operation that changes the mode. In cases where the mode changes while PacketEn=0, the processor will send out a MODE.Exec along with the TIP.PGE when tracing resumes. The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that from the last MODE.Exec packet, if there was no PSB in between.</td></tr>
<tr><td>Application</td><td colspan="2">MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.

MODE.TSX Packet

Table 35-27. MODE.TSX Packet Definition

<table>
<tr><td>Name</td><td colspan="3">MODE.TSX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	0	0	1	0	0	0	TXAbort	InTX

</td></tr>
<tr><td>Dependencies</td><td>TriggerEn and ContextEn</td><td>Generation Scenario</td><td>XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if INTX changes, Asynchronous TSX Abort, PSB+</td></tr>
<tr><td>Description</td><td colspan="3">Indicates when a TSX transaction (either HLE or RTM) begins, commits, or aborts. Instructions executed transactionally will be “rolled back” if the transaction is aborted.

TXAbort	InTX	Implication
1	1	N/A
0	1	Transaction begins, or executing transactionally
1	0	Transaction aborted
0	0	Transaction committed, or not executing transactionally

</td></tr>
<tr><td>Application</td><td colspan="3">If PacketEn=1, MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP.
MODE.TSX packets may be generated when PacketEn=0, due to FilterEn=0. In this case, only the last MODE.TSX generated before TIP.PGE need be applied.</td></tr>
</table>

35.4.2.9 TraceStop Packet

Table 35-28. TraceStop Packet Definition

<table>
<tr><td>Name</td><td colspan="3">TraceStop Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn</td><td>Generation Scenario</td><td>Taken branch with target in TraceStop IP region, MOV CR3 in TraceStop IP region, or WRMSR that sets TraceEn in TraceStop IP region.</td></tr>
<tr><td>Description</td><td colspan="3">Indicates when software has entered a user-configured TraceStop region.
When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated.
The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 35.2.4.3 for more details.</td></tr>
<tr><td>Application</td><td colspan="3">.If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any).
If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.</td></tr>
</table>

35.4.2.10 Core:Bus Ratio (CBR) Packet

Table 35-29. CBR Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Core:Bus Ratio (CBR) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="8">Core:Bus Ratio</td></tr>
<tr><td>3</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td>After any frequency change, on C-state wake up, PSB+, and after enabling trace packet generation.</td></tr>
<tr><td>Description</td><td colspan="3">Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.</td></tr>
<tr><td>Application</td><td colspan="3">All packets following the CBR represent instructions that executed with the new core:bus ratio, while all preceding packets (aside from timing packets) represent instructions that executed with the prior ratio. There is not a precise IP provided, to which to bind the CBR packet.</td></tr>
</table>

35.4.2.11 Timestamp Counter (TSC) Packet

Table 35-30. TSC Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Timestamp Counter (TSC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">SW TSC[7:0]</td></tr>
<tr><td>2</td><td colspan="8">SW TSC[15:8]</td></tr>
<tr><td>3</td><td colspan="8">SW TSC[23:16]</td></tr>
<tr><td>4</td><td colspan="8">SW TSC[31:24]</td></tr>
<tr><td>5</td><td colspan="8">SW TSC[39:32]</td></tr>
<tr><td>6</td><td colspan="8">SW TSC[47:40]</td></tr>
<tr><td>7</td><td colspan="8">SW TSC[55:48]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td>Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop, This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.</td></tr>
<tr><td>Description</td><td colspan="3">When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.</td></tr>
<tr><td>Application</td><td colspan="3">TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.</td></tr>
</table>

35.4.2.12 Mini Time Counter (MTC) Packet

Table 35-31. MTC Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Mini time Counter (MTC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3"><table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">CTC[N+7:N]</td></tr>
</table></td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && TriggerEn</td><td>Generation Scenario</td><td>Periodic, based on the core crystal clock, or Always Running Timer (ART).</td></tr>
<tr><td>Description</td><td colspan="3">When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to (ART >> N) & 0xFF. The frequency of the ART is related to the Maximum Non-Turbo frequency, and the ratio can be determined from CPUID leaf 15H, as described in Section 35.8.3.
Software can select the threshold N, which determines the MTC frequency by setting the IA32_RTIT_CTL.MTCFreq field (see Section 35.2.7.2) to a supported value using the lookup enumerated by CPUID (see Section 35.3.1).
See Section 35.8.3 for details on how to use the MTC payload to track TSC time.
MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.
There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that >256 consecutive MTC packets should ever be dropped.</td></tr>
<tr><td>Application</td><td colspan="3">MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.</td></tr>
</table>

35.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 35-32. TMA Packet Definition

<table>
<tr><td>Name</td><td colspan="3">TSC/MTC Alignment (TMA) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="8">CTC[7:0]</td></tr>
<tr><td>3</td><td colspan="8">CTC[15:8]</td></tr>
<tr><td>4</td><td colspan="7">Reserved</td><td>0</td></tr>
<tr><td>5</td><td colspan="8">FastCounter[7:0]</td></tr>
<tr><td>6</td><td colspan="7">Reserved</td><td>FC[8]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td>Sent with any TSC packet.</td></tr>
<tr><td>Description</td><td colspan="3">The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 35.8.3.2 for details on how to make this calculation.</td></tr>
<tr><td>Application</td><td colspan="3">TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.</td></tr>
</table>

35.4.2.14 Cycle Count Packet (CYC) Packet

Table 35-33. Cycle Count Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Cycle Count (CYC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="5">Cycle Counter[4:0]</td><td>Exp</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td colspan="7">Cycle Counter[11:5]</td><td>Exp</td></tr>
<tr><td>2</td><td colspan="7">Cycle Counter[18:12]</td><td>Exp</td></tr>
<tr><td>...</td><td colspan="8">... (if Exp = 1 in the previous byte)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.CYCEn && TriggerEn</td><td>Generation Scenario</td><td>Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 35.3.6 for CYC-eligible packets.</td></tr>
<tr><td>Description</td><td colspan="3">The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length.
If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on.
CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 35.2.7.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 35.3.6.
When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow.
In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor's internal cycle counter doesn't cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen.
When CycThresh>0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone.
When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time.</td></tr>
<tr><td>Application</td><td colspan="3">CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet.
If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet.</td></tr>
</table>

35.4.2.15 VMCS Packet

Table 35-34. VMCS Packet Definition

<table>
<tr><td>Name</td><td colspan="3">VMCS Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">VMCS Base Address [19:12]</td></tr>
<tr><td>3</td><td colspan="8">VMCS Base Address [27:20]</td></tr>
<tr><td>4</td><td colspan="8">VMCS Base Address [35:28]</td></tr>
<tr><td>5</td><td colspan="8">VMCS Base Address [43:36]</td></tr>
<tr><td>6</td><td colspan="8">VMCS Base Address [51:44]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn;
Also in VMX operation.</td><td>Generation Scenario</td><td>Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 35.5).</td></tr>
<tr><td>Description</td><td colspan="3">The VMCS packet provides an address related to a VMCS pointer for a decoder to determine the transition of code contexts:
• On a successful VMPTRLD (i.e., a VMPTRLD that doesn't fault, fail, or VM exit), the VMCS packet contains the address of the current working VMCS pointer of the logical processor that will execute a VM guest context.
• On SMM VM exits, the VMCS packet provides the STM VMCS base address (SMM Transfer VMCS pointer), if VMCS-based controls are clear (see Section 35.5.1). See Section 35.6 on tracing inside and outside STM.
• On VM entries that return from SMM, the VMCS packet provides the current working VMCS pointer of the guest VM (see Section 35.6), if VMCS-based controls are clear (see Section 35.5.1). Root versus Non-Root operation can be distinguished from the PIP.NR bit.
If a VMCS packet is generated before a VMCS has been loaded, or after it has been cleared, the base address value will be all 1s.
VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.</td></tr>
<tr><td>Application</td><td colspan="3">The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique.
When a VMCS is encountered, a decoder should do the following:
▪ If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this VMCS is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new VMCS base pointer value to the TIP payload IP.
▪ Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

35.4.2.16 Overflow (OVF) Packet

Table 35-35. OVF Packet Definition

Name	Overflow (OVF) Packet		
Packet Format	(see packet format table below)		
Dependencies	TriggerEn	Generation Scenario	On resolution of internal buffer overflow
Description	OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. If BranchEN= 1, OVF is followed by a FUP or TIP.PGE which will provide the IP at which packet generation resumes. See Section 35.3.8.		
Application	When an OVF packet is encountered, the decoder should skip to the IP given in the subsequent FUP or TIP.PGE. The cycle counter for the CYC packet will be reset at the time the OVF packet is sent. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that preceded the overflow.		

Packet Format:

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	1	1	1	0	0	1	1

35.4.2.17 Packet Stream Boundary (PSB) Packet

Table 35-36. PSB Packet Definition

Name	Packet Stream Boundary (PSB) Packet
Packet Format	(see packet format table below)

Packet Format:

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0
5	1	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
7	1	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0
9	1	0	0	0	0	0	1	0
10	0	0	0	0	0	0	1	0
11	1	0	0	0	0	0	1	0
12	0	0	0	0	0	0	1	0
13	1	0	0	0	0	0	1	0
14	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	1	0

Table 35-36. PSB Packet Definition (Contd.)

Dependencies	TriggerEn	Generation Scenario	Periodic, based on the number of output bytes generated while tracing. PSB is sent when IA32_RTIT_STATUS.PacketByteCnt=0, and each time it crosses the software selected threshold after that. May be sent for other micro-architectural conditions as well.
Description	PSB is a unique pattern in the packet output log, and hence serves as a sync point for the decoder. It is a pattern that the decoder can search for in order to get aligned on packet boundaries. This packet is periodic, based on the number of output bytes, as indicated by IA32_RTIT_STATUS.PacketByteCnt. The period is chosen by software, via IA32_RTIT_CTL.PSBFreq (see Section 35.2.7.2). Note, however, that the PSB period is not precise, it simply reflects the average number of output bytes that should pass between PSBs. The processor will make a best effort to insert PSB as quickly after the selected threshold is reached as possible. The processor also may send extra PSB packets for some micro-architectural conditions. PSB also serves as the leading packet for a set of "status-only" packets collectively known as PSB+ (Section 35.3.7).		
Application	When a PSB is seen, the decoder should interpret all following packets as "status only", until either a PSBEND or OVF packet is encountered. "Status only" implies that the binding and ordering rules to which these packets normally adhere are ignored, and the state they carry can instead be applied to the IP payload in the FUP packet that is included.		

35.4.2.18 PSBEND Packet

Table 35-37. PSBEND Packet Definition

Name	PSBEND Packet		
Packet Format	(see below)		
Dependencies	TriggerEn	Generation Scenario	Always follows PSB packet, separated by PSB+ packets
Description	PSBEND is simply a terminator for the series of "status only" (PSB+) packets that follow PSB (Section 35.3.7).		
Application	When a PSBEND packet is seen, the decoder should cease to treat packets as "status only".		

Packet Format:

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	1	1

35.4.2.19 Maintenance (MNT) Packet

Table 35-38. MNT Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Maintenance (MNT) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	1	1
2	1	0	0	0	1	0	0	0
3	Payload[7:0]							
4	Payload[15:8]							
5	Payload[23:16]							
6	Payload[31:24]							
7	Payload[39:32]							
8	Payload[47:40]							
9	Payload[55:48]							
10	Payload[63:56]							

</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td>Implementation specific.</td></tr>
<tr><td>Description</td><td colspan="3">This packet is generated by hardware, the payload meaning is model-specific.</td></tr>
<tr><td>Application</td><td colspan="3">Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.</td></tr>
</table>

35.4.2.20 PAD Packet

Table 35-39. PAD Packet Definition

<table>
<tr><td>Name</td><td colspan="3">PAD Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0

</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td>Implementation specific</td></tr>
<tr><td>Description</td><td colspan="3">PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.</td></tr>
<tr><td>Application</td><td colspan="3">Ignore PAD packets.</td></tr>
</table>

35.4.2.21 PTWRITE Packet

Table 35-40. PTW Packet Definition

<table>
<tr><td>Name</td><td colspan="3">PTW Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td colspan="2">PayloadBytes</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">Payload[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Payload[15:8]</td></tr>
<tr><td>4</td><td colspan="8">Payload[23:16]</td></tr>
<tr><td>5</td><td colspan="8">Payload[31:24]</td></tr>
<tr><td>6</td><td colspan="8">Payload[39:32]</td></tr>
<tr><td>7</td><td colspan="8">Payload[47:40]</td></tr>
<tr><td>8</td><td colspan="8">Payload[55:48]</td></tr>
<tr><td>9</td><td colspan="8">Payload[63:56]</td></tr>
</table>
</td></tr>
<tr><td></td><td colspan="3">The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:
<table>
<tr><th>PayloadBytes</th><th>Bytes of Payload</th></tr>
<tr><td>‘00</td><td>4</td></tr>
<tr><td>‘01</td><td>8</td></tr>
<tr><td>‘10</td><td>Reserved</td></tr>
<tr><td>‘11</td><td>Reserved</td></tr>
</table>
IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn & FilterEn & PTWEn</td><td>Generation Scenario</td><td>PTWRITE Instruction</td></tr>
<tr><td>Description</td><td colspan="3">Contains the value held in the PTWRITE operand.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.</td></tr>
</table>

35.4.2.22 Execution Stop (EXSTOP) Packet

Table 35-41. EXSTOP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">EXSTOP Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
IP bit indicates if a FUP will follow.</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td>C-state entry, P-state change, or other processor clock power-down. Includes :
• Entry to C-state deeper than C0.0
• TM1/2
• STPCLK#
• Frequency change due to IA32_CLOCK_MODULATION, Turbo</td></tr>
<tr><td>Description</td><td colspan="3">This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes.
If EXSTOP is generated while ContextEn is set, the IP bit will be set, and EXSTOP will be followed by a FUP packet containing the IP at which execution stopped. More precisely, this will be the IP of the oldest instruction that has not yet completed.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">If a FUP follows EXSTOP (hence IP bit set), the EXSTOP can be bound to the FUP IP. Otherwise the IP is not known.
Time of powerdown can be inferred from the preceding CYC, if CYCEn=1. Combined with the TSC at the time of wake (if TSCEn=1), this can be used to determine the duration of the powerdown.</td></tr>
</table>

35.4.2.23 MWAIT Packet

Table 35-42. MWAIT Packet Definition

<table>
<tr><td>Name</td><td colspan="3">MWAIT Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">MWAIT Hints[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Reserved</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="6">Reserved</td><td colspan="2">EXT[1:0]</td></tr>
<tr><td>7</td><td colspan="8">Reserved</td></tr>
<tr><td>8</td><td colspan="8">Reserved</td></tr>
<tr><td>9</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn & ContextEn</td><td>Generation Scenario</td><td>MWAIT instruction, or I/O redirection to MWAIT, that complete without fault or VMexit.</td></tr>
<tr><td>Description</td><td colspan="3">Indicates that an MWAIT operation to C-state deeper than C0.0 completed. The MWAIT hints and extensions passed in by software are exposed in the payload.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">The MWAIT packet should bind to the IP of the next FUP, which will be the IP of the instruction that caused the MWAIT. This FUP will be shared with EXSTOP.</td></tr>
</table>

35.4.2.24 Power Entry (PWRE) Packet

Table 35-43. PWRE Packet Definition

<table>
<tr><td>Name</td><td colspan="3">PWRE Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td>HW</td><td colspan="7">Reserved</td></tr>
<tr><td>3</td><td colspan="4">Resolved Thread C-State</td><td colspan="4">Resolved Thread Sub C-State</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td>Transition to a C-state deeper than C0.0.</td></tr>
<tr><td>Description</td><td colspan="3">Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.
Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core.
If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, HLT, or shut-down), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 14.5, “Hardware Duty Cycling (HDC)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B) is an example of such a case.</td></tr>
<tr><td>Application</td><td colspan="3">When transitioning from C0.0 to a deeper C-state, the PWRE packet will be followed by an EXSTOP. If that EXSTOP packet has the IP bit set, then the following FUP will provide the IP at which the C-state entry occurred. Subsequent PWRE packets generated before the next PWRX should bind to the same IP.</td></tr>
</table>

35.4.2.25 Power Exit (PWRX) Packet

Table 35-44. PWRX Packet Definition

<table>
<tr><td>Name</td><td colspan="3">PWRX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="4">Last Core C-State</td><td colspan="4">Deepest Core C-State</td></tr>
<tr><td>3</td><td colspan="4">Reserved</td><td colspan="4">Wake Reason</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td>Transition from a C-state deeper than C0.0 to C0.</td></tr>
<tr><td>Description</td><td colspan="3">Indicates processor return to thread C0 from a C-state deeper than C0.0.
The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested.
The Wake Reason field is one-hot, encoded as follows:
<table>
<tr><th>Bit</th><th>Field</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt</td><td>Wake due to external interrupt received.</td></tr>
<tr><td>1</td><td>Reserved</td><td></td></tr>
<tr><td>2</td><td>Store to Monitored Address</td><td>Wake due to store to monitored address.</td></tr>
<tr><td>3</td><td>HW Wake</td><td>Wake due to hardware autonomous condition, such as HDC.</td></tr>
</table>
</td></tr>
<tr><td>Application</td><td colspan="3">PWRX will always apply to the same IP as the PWRE. The time of wake can be discerned from (optional) timing packets that precede PWRX.</td></tr>
</table>

35.5 TRACING IN VMX OPERATION

On processors that IA32_VMX_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. A series of mechanisms exist to allow the VMM to configure tracing based on the desired trace domain, and on the consumer of the trace output. The VMM can configure specific VM-execution controls to control what virtualization-specific data are included within the trace packets (see Section 35.5.1 for details). MSR save and load lists can be employed by the VMM to restrict tracing to the desired context (see Section 35.5.2 for details). These configuration options are summarized in Table 35-45. Table 35-45 covers common Intel PT usages while SMIs are handled by the default SMM treatment. Tracing with SMM Transfer Monitor is described in Section 35.6.

Table 35-45. Common Usages of Intel PT and VMX

Target Domain	Output Consumer	Virtualize Output	Configure VMCS Controls	TraceEN Configuration	Save/Restore MSR states of Trace Configuration
System-Wide (VMM + VMs)	Host	NA	Default Setting (no suppression)	WRMSR or XRSTORS by Host	NA
VMM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to disable tracing in VM, enable tracing on VM exits	NA
VM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	NA
Intel PT Aware Guest(s)	Per Guest	VMM adds trace output virtualization	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	VMM Update guest state on XRSTORS-exiting VM exits

35.5.1 VMX-Specific Packets and VMCS Controls

In all of the usages of VMX and Intel PT, the decoder in the host or VMM context can identify the occurrences of VMX transitions with the aid of VMX-specific packets. Packets relevant to VMX fall into the follow two kinds:

- VMCS Packet: The VMX transitions of individual VM can be distinguished by a decoder using the base address field in a VMCS packet. The base address field stores the VMCS pointer address of a successful VMPTRLD. A VMCS packet is sent on a successful execution of VMPTRLD. See Section 35.4.2.15 for details.
- NonRoot (NR) bit field in PIP packet: PIP packets are generated with each VM entry/exit. The NR bit in a PIP packet is set when in VMX non-Root operation. Thus a transition of the NR bit from 0 to 1 indicates the occurrence of a VM entry, and a transition of 1 to 0 indicates the occurrence of a VM exit.

Processors with IA32_VMX_MISC[bit 14]= 1 also provides VMCS controls that a VMM can configure to prevent VMX-specific information from leaking across virtualization boundaries.

Table 35-46. VMCS Controls For Intel Processor Trace

Name	Type	Bit Position	Value	Behavior
Conceal VMX non-root operation from Intel PT	VM-execution control	19	0	PIPs generated in VM non-root operation will set the PIP.NR bit. PSB+ in VMX non-root operation will include the VMCS packet, to ensure that the decoder knows which guest is currently in use.
			1	PIPs generated in VMX non-root operation will not set the PIP.NR bit. PSB+ in VMX non-root operation will not include the VMCS packet.
Conceal VM exits from Intel PT	VM-exit control	24	0	PIPs are generated on VM exit, with NonRoot=0. On VM exit to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM exit, and no VMCS packet is generated on VM exit to SMM.
Conceal VM entries from Intel PT	VM-entry control	17	0	PIPs are generated on VM entry, with NonRoot=1 if the destination of the VM entry is VMX non-root operation. On VM entry to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM entry, and no VMCS packet is generated on VM entry to SMM.

The default setting for the VMCS controls that interacts with Intel PT is to enable all VMX-specific packet information. The scenarios that would use the default setting also do not require the VMM to use MSR load list to manage the configuration of turning-on/off of trace packet generation across VM exits.

If IA32_VMX_MISC[bit 14] reports 0, any attempt to set the VMCS control bits in Table 35-46 will result in a failure on guest entry.

35.5.2 Managing Trace Packet Generation Across VMX Transitions

In tracing scenarios that collect packets for both VMX root operation and VMX non-root operation, a host executive can manage the MSRs associated with trace packet generation directly. The states of these MSRs need not be modified using MSR load list or MSR save list across VMX transitions.

For tracing scenarios that collect only packets within either VMX root operation or VMX non-root operation, the VMM can use the MSR load list and/or MSR save list to toggle IA32_RTIT_CTL.TraceEn.

35.5.2.1 System-Wide Tracing

When a host or VMM configures Intel PT to collect trace packets of the entire system, it can leave the VMCS controls clear to allow VMX-specific packets to provide information across VMX transitions. MSR load list is not used across VM exits or VM entries, nor is VM-exit MSR save list.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 35-47.

Table 35-47. Packets on VMX Transitions (System-Wide Tracing)

Event	Packets	Description
VM exit	FUP(GuestIP)	The FUP indicates at which point in the guest flow the VM exit occurred. This is important, since VM exit can be an asynchronous event. The IP will match that written into the VMCS.
	PIP(HostCR3, NR=0)	The PIP packet provides the new host CR3 value, as well as indication that the logical processor is entering VMX root operation. This allows the decoder to identify the change of executing context from guest to host and load the appropriate set of binaries to continue decode.
	TIP(HostIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	PIP(GuestCR3, NR=1)	The PIP packet provides the new guest CR3 value, as well as indication that the logical processor is entering VMX non-root operation. This allows the decoder to identify the change of executing context from host to guest and load the appropriate set of binaries to continue decode.
	TIP(GuestIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX non-root operation. This should match the IP value read out from the VMCS. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.

Since the packet suppression controls are cleared, the VMCS packet will be included in all PSB+ for this usage scenario. Thus the decoder can distinguish the execution context of different VMs. Additionally, it will be generated on VMPTRLD. Thus the decoder can distinguish the execution context of different VMs.

When the host VMM configures a system to collect trace packets in this scenario, it should emulate CPUID to report CPUID.(EAX=07H, ECX=0):EBX[bit 26] with 0 to guests, indicating to guests that Intel PT is not available.

VMX TSC Manipulation

The TSC packets generated while in VMX non-root operation will include any changes resulting from the use of a VMM's use of the TSC offsetting or TSC scaling VMCS control (see Chapter 25, "VMX Non-Root Operation"). In this system-wide usage model, the decoder may need to account for the effect of per-VM adjustments in the TSC packets generated in VMX non-root operation and the absence of TSC adjustments in TSC packets generated in VMX root operation. The VMM can supply this information to the decoder.

35.5.2.2 Host-Only Tracing

When trace packets in VMX non-root operation are not desired, the VMM can use VM-entry MSR load list with IA32_RTIT_CTL.TraceEn=0 to disable trace packet generation in guests, set IA32_RTIT_CTL.TraceEn=1 via VM-exit MSR load list.

When tracing only the host, the decoder does not need information about the guests, the VMCS controls for suppressing VMX-specific packets can be set to reduce the packets generated. VMCS packets will still be generated on successful VMPTRLD and in PSB+ generated in the Host, but these will be unused by the decoder.

The packets of interests to a decoder when trace packets are collected for host-only tracing are shown in Table 35-48.

Table 35-48. Packets on VMX Transitions (Host-Only Tracing)

Event	Packets	Description
VM exit	TIP.PGE(HostIP)	The TIP.PGE indicates that trace packet generation is enabled and gives the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	TIP.PGD()	The TIP indicates that trace packet generation was disabled. This ensure that all buffered packets are flushed out.

35.5.2.3 Guest-Only Tracing

A VMM can configure trace packet generation while in non-root operation for guests executing normally. This is accomplished by utilizing the MSR load lists across VM exit and VM entry to confine trace packet generation to stay within the guest environment.

For this usage, the VM-entry MSR load list is programmed to turn on trace packet generation. The VM-exit MSR load list is used to clear TraceEn=0 to disable trace packet generation in the host. Further, if it is preferred that the guest packet stream contain no indication that execution was in VMX non-root operation, the VMM should set the VMCS controls described in Table 35-46.

35.5.2.4 Virtualization of Guest Output Packet Streams

Each Intel PT aware guest OS can produce one or more output packet streams to destination addresses specified as guest physical address (GPA) using context-switched IA32_RTIT_OUTPUT_BASE within the guest. The processor generates trace packets to the platform physical address specified in IA32_RTIT_OUTPUT_BASE, and those specified in the ToPA tables. Thus, a VMM that supports Intel PT aware guest OS may wish to virtualize the output configurations of IA32_RTIT_OUTPUT_BASE and ToPA for each trace configuration state of all the guests.

35.5.2.5 Emulation of Intel PT Traced State

If a VMM emulates an element of processor state by taking a VM exit on reads and/or writes to that piece of state, and the state element impacts Intel PT packet generation or values, it may be incumbent upon the VMM to insert or modify the output trace data.

If a VM exit is taken on a guest write to CR3 (including “MOV CR3” as well as task switches), the PIP packet normally generated on the CR3 write will be missing.

To avoid decoder confusion when the guest trace is decoded, the VMM should emulate the missing PIP by writing it into the guest output buffer. If the guest CR3 value is manipulated, the VMM may also need to manipulate the IA32_RTIT_CR3_MATCH value, in order to ensure the trace behavior matches the guest's expectation.

Similarly, if a VMM emulates the TSC value by taking a VM exit on RDTSC, the TSC packets generated in the trace may mismatch the TSC values returned by the VMM on RDTSC. To ensure that the trace can be properly aligned with software logs based on RDTSC, the VMM should either make corresponding modifications to the TSC packet values in the guest trace, or use mechanisms such as TSC offsetting or TSC scaling in place of exiting.

35.5.2.6 TSC Scaling

When TSC scaling is enabled for a guest using Intel PT, the VMM should ensure that the value of Maximum Non-Turbo Ratio[15:8] in MSR_PLATFORM_INFO (MSR 0CEH) and the TSC/”core crystal clock” ratio (EBX/EAX) in CPUID leaf 15H are set in a manner consistent with the resulting TSC rate that will be visible to the VM. This will allow the decoder to properly apply TSC packets, MTC packets (based on the core crystal clock or ART, whose frequency is indicated by CPUID leaf 15H), and CBR packets (which indicate the ratio of the processor frequency to the Max

Non-Turbo frequency). Absent this, or separate indication of the scaling factor, the decoder will be unable to properly track time in the trace. See Section 35.8.3 for details on tracking time within an Intel PT trace.

35.5.2.7 Failed VM Entry

The packets generated by a failed VM entry depend both on the VMCS configuration, as well as on the type of failure. The results to expect are summarized in the table below. Note that packets in *italics* may or may not be generated, depending on implementation choice, and the point of failure.

Table 35-49. Packets on a Failed VM Entry

Usage Model	Entry Configuration	Early Failure (fall through to Next IP)	Late Failure (VM exit)
System-Wide	No MSR load list	TIP (NextIP)	*PIP(Guest CR3, NR=1), TraceEn 0->1 Packets (See Section 35.2.7.3),* PIP(HostCR3, NR=0), TIP(HostIP)
VMM Only	MSR load list disables TraceEn	TIP (NextIP)	*TraceEn 0->1 Packets (See Section 35.2.7.3),* TIP(HostIP)
VM Only	MSR load list Enables TraceEn	None	None

35.5.2.8 VMX Abort

VMX abort conditions take the processor into a shutdown state. On a VM exit that leads to VMX abort, some packets (FUP, PIP) may be generated, but any expected TIP, TIP.PGE, or TIP.PGD may be dropped.

35.6 TRACING AND SMM TRANSFER MONITOR (STM)

SMM Transfer Monitor is a VMM that operates inside SMM while in VMX root operation. An STM operates in conjunction with an executive monitor. The latter operates outside SMM and in VMX root operation. Transitions from the executive monitor or its VMs to the STM are called SMM VM exits. The STM returns from SMM via a VM entry to the VM in VMX non-root operation or the executive monitor in VMX root operation.

Intel PT supports tracing in an STM similar to tracing support for VMX operation as described above in Section 35.7. As a result, on a SMM VM exit resulting from #SMI, TraceEn is not saved and then cleared. Software can save the state of the trace configuration MSRs and clear TraceEn using the MSR load/save lists.

35.7 PACKET GENERATION SCENARIOS

Table 35-50 and Table 35-52 illustrate the packets generated in various scenarios. In the heading row, PacketEn is abbreviated as PktEn, ContextEn as CntxEn. Note that this assumes that TraceEn=1 in IA32_RTIT_CTL, while TriggerEn=1 and Error=0 in IA32_RTIT_STATUS, unless otherwise specified. Entries that do not matter in packet generation are marked "D.C." Packets followed by a "?" imply that these packets depend on additional factors, which are listed in the "Other Dependencies" column.

In Table 35-50, PktEn is evaluated based on TiggerEn & ContextEn & FilterEn & BranchEn.

Table 35-50. Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
1a	Normal non-jump operation	0	0	D.C.		None
1b	Normal non-jump operation	1	1	1		None
2a	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
2b	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	PSB, PSBEND (see Section 35.4.2.17)
2d	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	1	1	TSC if TSCEn=1; TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, MODE.Exec, TIP.PGE(NLIP)
2e	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	1	1		MODE.Exec, TIP.PGE(NLIP), PSB, PSBEND (see Section 35.4.2.8, 35.4.2.7, 35.4.2.13,35.4.2.15, 35.4.2.17)
3a	WRMSR that changes TraceEn 1 -> 0	0	0	D.C.		None
3b	WRMSR that changes TraceEn 1 -> 0	1	0	D.C.		FUP(CLIP), TIP.PGD()
5a	MOV to CR3	0	0	0		None
5f	MOV to CR3	0	0	1	TraceStop if executed in a TraceStop region	PIP(NewCR3,NR?), TraceStop?
5b	MOV to CR3	0	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?), MODE.Exec?, TIP.PGE(NLIP)
5c	MOV to CR3	1	0	0		TIP.PGD()
5e	MOV to CR3	1	0	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *TraceStop if executed in a TraceStop region	PIP(NewCR3, NR?), TIP.PGE(NLIP), TraceStop?
5d	MOV to CR3	1	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0	PIP(NewCR3, NR?)
6a	Unconditional direct near jump	0	0	D.C.		None
6b	Unconditional direct near jump	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
6c	Unconditional direct near jump	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
6d	Unconditional direct near jump	1	1	1		None
7a	Conditional taken jump or compressed RET that does not fill up the internal TNT buffer	0	0	D.C.		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
7b	Conditional taken jump or compressed RET	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
7e	Conditional taken jump or compressed RET, with empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(), TraceStop?
7f	Conditional taken jump or compressed RET, with non-empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TNT, TIP.PGD(), TraceS-top?
7d	Conditional taken jump or compressed RET that fills up the internal TNT buf-fer	1	1	1		TNT
8a	Conditional non-taken jump	0	0	D.C.		None
8d	Conditional not-taken jump that fills up the internal TNT buffer	1	1	1		TNT
9a	Near indirect jump (JMP, CALL, or uncompressed RET)	0	0	D.C.		None
9b	Near indirect jump (JMP, CALL, or uncompressed RET)	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
9c	Near indirect jump (JMP, CALL, or uncompressed RET)	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
9d	Near indirect jump (JMP, CALL, or uncompressed RET)	1	1	1		TIP(BLIP)
10a	Far Branch (CALL/JMP/RET)	0	0	0		None
10f	Far Branch (CALL/JMP/RET)	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), Trace-Stop?
10b	Far Branch (CALL/JMP/RET)	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
10c	Far Branch (CALL/JMP/RET)	1	0	0		TIP.PGD()

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
10d	Far Branch (CALL/JMP/RET)	1	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and “Conceal VMX non-root operation from Intel PT” execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TIP.PGD(BLIP), TraceStop?
10e	Far Branch (CALL/JMP/RET)	1	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and “Conceal VMX non-root operation from Intel PT” execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
11a	HW Interrupt	0	0	0		None
11f	HW Interrupt	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and “Conceal VMX non-root operation from Intel PT” execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
11b	HW Interrupt	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and “Conceal VMX non-root operation from Intel PT” execution control = 0; * MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
11c	HW Interrupt	1	0	0		FUP(NLIP), TIP.PGD()
11d	HW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and “Conceal VMX non-root operation from Intel PT” execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(NLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
11e	HW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(NLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
12a	SW Interrupt	0	0	0		None
12f	SW Interrupt	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
12b	SW Interrupt	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
12c	SW Interrupt	1	0	0		FUP(CLIP), TIP.PGD()
12d	SW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
12e	SW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
13a	Exception/Fault	0	0	0		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
13f	Exception/Fault	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
13b	Exception/Fault	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
13c	Exception/Fault	1	0	0		FUP(CLIP), TIP.PGD()
13d	Exception/Fault	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
13e	Exception/Fault	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
14a	SMI (TraceEn cleared)	0	0	D.C.		None
14b	SMI (TraceEn cleared)	1	0	0		FUP(SMRAM,LIP), TIP.PGD()
14f	SMI (TraceEn cleared)	1	0	1		NA
14c	SMI (TraceEn cleared)	1	1	1		NA
15a	RSM, TraceEn restored to 0	0	0	0		None
15b	RSM, TraceEn restored to 1	0	0	D.C.		See WRMSR cases for packets on enable

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
15c	RSM, TraceEn restored to 1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15e	RSM (TraceEn=1, goes to shutdown)	1	0	0		None
15f	RSM (TraceEn=1, goes to shutdown)	1	0	1		None
15d	RSM (TraceEn=1, goes to shutdown)	1	1	1		None
16i	Vmext	0	0	0		None
16a	Vmext	0	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSh.LIP is in a TraceStop region	PIP(HostCR3, NR=0)?, TraceStop?
16b	VM exit, MSR list sets TraceEn=1	0	0	0		See WRMSR cases for packets on enable. FUP IP is VMCSh.LIP
16c	VM exit, MSR list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSh.LIP
16e	VM exit	0	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(HostCR3, NR=0)?, MODE.Exec?, TIP.PGE(VMCSh.LIP)
16f	VM exit, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0;	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, TIP.PGD
16j	VM exit, ContextEN 1->0	1	0	0		FUP(VMCSg.LIP), TIP.PGD
16g	VM exit	1	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSh.LIP is in a TraceStop region	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, TIP.PGD(VMCSh.LIP), TraceStop?
16h	VM exit	1	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, MODE.Exec, TIP(VMCSh.LIP)
17a	VM entry	0	0	0		None
17b	VM entry	0	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSg.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TraceStop?
17c	VM entry, MSR load list sets TraceEn=1	0	0	1		See WRMSR cases for packets on enable. FUP IP is VMCSg.LIP

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
17d	VM entry, MSR load list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSg.LIP
17f	VM entry, FilterEN 0->1	0	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec?, TIP.PGE(VMCSg.LIP)
17j	VM entry, ContextEN 0->1	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec, TIP.PGE(VMCSg.LIP)
17g	VM entry, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0;	PIP(GuestCR3, NR=1)?, TIP.PGD
17h	VM entry	1	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSg.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TIP.PGD(VMCSg.LIP), TraceStop?
17i	VM entry	1	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec, TIP(VMCSg.LIP)
20a	EENTER/ERESUME to non-debug enclave	0	0	0		None
20c	EENTER/ERESUME to non-debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
21a	EEXIT from non-debug enclave	0	0	D.C.		None
21b	EEXIT from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
22a	AEX/EEE from non-debug enclave	0	0	D.C.		None
22b	AEX/EEE from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
23a	EENTER/ERESUME to debug enclave	0	0	D.C.		None
23b	EENTER/ERESUME to debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
23c	EENTER/ERESUME to debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
23d	EENTER/ERESUME to debug enclave	0	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?
23e	EENTER/ERESUME to debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
24f	EEXIT from debug enclave	0	0	D.C.		None
24b	EEXIT from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
24d	EEXIT from debug enclave	1	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
24e	EEXIT from debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
25a	AEX/EEE from debug enclave	0	0	D.C.		None
25b	AEX/EEE from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
25d	AEX/EEE from debug enclave	1	0	1	*For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), TIP.PGD(AEP.LIP)
25e	AEX/EEE from debug enclave	1	1	1	*MODE.Exec if the value is different, since last TIP.PGD *For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), MODE.Exec?, TIP(AEP.LIP)
26a	XBEGIN/XACQUIRE	0	0	D.C.		None
26d	XBEGIN/XACQUIRE that does not set InTX	1	1	1		None
26e	XBEGIN/XACQUIRE that sets InTX	1	1	1		MODE(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0	D.C.		None
27d	XEND/XRELEASE that does not clear InTX	1	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1	1		MODE(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TraceStop?
28b	XABORT(Async XAbort, or other)	0	1	1		MODE(InTX=0, TXAbort=1), TIP.PGE(BLIP)
28c	XABORT(Async XAbort, or other)	1	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	BIP(0), TraceStop?
30c	INIT (BSP)	0	1	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, PIP(0), TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0	0		FUP(NLIP), TIP.PGD()
30e	INIT (BSP)	1	0	1	* PIP if OS=1 *TraceStop if RESET.LIP is in a TraceStop region	FUP(NLIP), PIP(0), TIP.PGD, TraceStop?

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
30f	INIT (BSP)	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB * PIP if OS=1	FUP(NLIP), PIP(0)?, MODE.Exec?, TIP(ResetLIP)
31a	INIT (AP, goes to wait-for-SIPI)	0	D.C.	D.C.		None
31b	INIT (AP, goes to wait-for-SIPI)	1	D.C.	D.C.	* PIP if OS=1	FUP(NLIP), PIP(0)
32a	SIPI	0	0	0		None
32c	SIPI	0	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(SIPI-LIP)
32d	SIPI	1	0	0		TIP.PGD
32e	SIPI	1	0	1	*TraceStop if SIPI LIP is in a TraceStop region	TIP.PGD(SIPILIP); TraceS-top?
32f	SIPI	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP(SIPILIP)
33a	MWAIT (to C0)	D.C.	D.C.	D.C.		None
33b	MWAIT (to higher-numbered C-State, packet sent on wake)	D.C.	D.C.	D.C.	*TSC if TSCEn=1 *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PwrEvtEn).

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.1	MWAIT or I/O redir to MWAIT, gets #UD or #GP fault	dc	dc	dc		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	dc	dc	dc		See VM exit examples (16[a-z] in Table 35-50) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests C0, or monitor not armed, or VMX virtual-interrupt delivery	dc	dc	dc		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	dc	1		MWAIT(Cy), PWRE(Cx), EXSTOP(IP), FUP(CLIP)
16.5a	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP, TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.5b	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	dc	1		MWAIT(5), PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.9a	HLT, Triple-fault shutdown, #MC with CR4.MCE=0, RSM to Cx (x>0)	dc	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	dc	dc			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	dc	0	0		See “VMX Abort” (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.10b	VMX abort	dc	dc	1		See “VMX Abort” (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	dc	0	0		See “RSM to Shutdown” (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.11b	RSM to Shutdown	dc	dc	1		See “RSM to Shutdown” (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.12a	INIT (BSP)	dc	0	0		See “INIT (BSP)” (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP
16.12b	INIT (BSP)	dc	dc	1		See “INIT (BSP)” (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP(IP), FUP(NLIP)

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.13a	INIT (AP, goes to Wait-for-SIPI)	dc	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.13b	INIT (AP, goes to Wait-for-SIPI)	dc	dc	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP, TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.14b	Hardware Duty Cycling (HDC)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXS-TOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	dc	0	0		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP
16.15b	VM entry to HLT or Shutdown	dc	dc	1		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STP-CLK#	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP, TSC?, TMA?, CBR
16.16b	EIST in C0, S1/TM1/TM2, or STP-CLK#	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	dc	dc	dc		None
16.18	INTR during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x1) See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.18	SMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See “HW Interrupt” (cases 14[a-z] in Table 35-50) for BranchEn packets that follow.
16.19	NMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See “HW Interrupt” (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.2	Store to monitored address during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x4)
16.22	#MC, IERR, TSC deadline timer expiration, or APIC counter under-flow during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PTWEn).

Table 35-52. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	dc	dc	dc		None
16.24b	PTWRITE rm32/64, no fault	dc	0	0		None
16.24d	PTWRITE rm32, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	dc	dc	dc		See "Exception/fault" (cases 13[a-z] in Table 35-50) for BranchEn packets.

35.8 SOFTWARE CONSIDERATIONS

35.8.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section , SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follows:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM
   - Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
   - For processors on which Intel PT and LBR use are mutually exclusive (see Section 35.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

35.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix "IA32_RTIT_" in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.

- Relinquish ownership of the trace configuration MSRs by clearing the "enabled bits" of those configuration MSRs.

35.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

- Processor timestamp counter

  This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] =0.

- Core crystal clock

  The ratio of the core crystal clock to timestamp counter frequency is known as P, and can calculating CPUID.15H:EBX[31:0] / CPUID.15H:EAX[31:0]. The frequency of the core crystal clock is fixed and lower than that of the timestamp counter. The periodic MTC packet is generated based on software-selected multiples of the crystal clock frequency. The MTC packet is expected to occur more frequently than the TSC packet.

- Processor core clock

  The processor core clock frequency can vary due to P-state and thermal conditions. The CYC packet provides elapsed time as measured in processor core clock cycles relative to the last CYC packet.

A decoder can use all or some combination of these packets to track time at different resolutions throughout the trace packets.

35.8.3.1 Time Domain Relationships

The three domains are related by the following formula:

```
TimeStampValue = (CoreCrystalClockValue * P) + AdjustedProcessorCycles + Software_Offset;
```

The CoreCrystalClockValue can provide the coarse-grained component of the TSC value. P, or the TSC/"core crystal clock" ratio, can be derived from CPUID leaf 15H, as described in Section 35.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.

The Software_Offsets component includes software offsets that are factored into the timestamp value, such as IA32_TSC_ADJUST.

35.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 35.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the CoreCrystalClockValue. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

$(CTC_B - CTC_A)$, where $CTC_i = MTC_i[15:8] << IA32_RTIT_CTL.MTCFreq$ and i = A, B.

The time from a TSC packet to the subsequent MTC packet can be calculated using the TMA packet that follows the TSC packet. The TMA packet provides both the crystal clock value (lower 16 bits, in the CTC field) and the AdjustedProcessorCycles value (in the FastCounter field) that can be used in the calculation of the corresponding core crystal clock value of the TSC packet.

When the next MTC after a pair of TSC/TMA is seen, the number of crystal clocks passed since the TSC packet can be calculated by subtracting the TMA.CTC value from the time indicated by the MTC_{Next} packet by

$CTC_{Delta}[15:0] = (CTC_{Next}[15:0] - TMA.CTC[15:0])$, where $CTC_{Next} = MTC_{Payload} << IA32_RTIT_CTL.MTCFreq$.

The TMA.FastCounter field provides the fractional component of the TSC packet into the next crystal clock cycle.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the adjusted_processor_cycles component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the core:bus ratio value given in the CBR packet, by multiplying the accumulated cycle count value by $P1/CBR_{payload}$.

A greater level of precision may be achieved by calculating the CPU clock frequency, see Section 35.8.3.4 below for a method to do so using Intel PT packets.

CYCs can be used to estimate time between TSCs even without MTCs, though this will likely result in a reduction in estimated TSC precision.

35.8.3.3 VMX TSC Manipulation

When software executes in non-Root operation, additional offset and scaling factors may be applied to the TSC value. These are optional, but may be enabled via VMCS controls on a per-VM basis. See Chapter 25, “VMX Non-Root Operation” for details on VMX TSC offsetting and TSC scaling.

Like the value returned by RDTSC, TSC packets will include these adjustments, but other timing packets (such as MTC, CYC, and CBR) are not impacted. In order to use the algorithm above to estimate the TSC value when TSC scaling is in use, it will be necessary for software to account for the scaling factor. See Section 35.5.2.6 for details.

35.8.3.4 Calculating Frequency with Intel PT

Because Intel PT can provide both wall-clock time and processor clock cycle time, it can be used to measure the processor core clock frequency. Either TSC or MTC packets can be used to track the wall-clock time. By using CYC packets to count the number of processor core cycles that pass in between a pair of wall-clock time packets, the ratio between processor core clock frequency and TSC frequency can be derived. If the P1 frequency is known, it can be applied to determine the CPU frequency. See Section 35.8.3.1 above for details on the relationship between TSC, MTC, and CYC.

18. Updates to Chapter 40, Volume 3D

Change bars show changes to Chapter 40 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Updates to the following instructions: ENCLS and ENCLU.

ENCLS—Execute an Enclave System Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CF ENCLS	NP	V/V	SGX1	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.3

Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the "enable ENCLS exiting" VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the "enable ENCLS exiting" VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation and the "enable ENCLS exiting" VM-execution control is 1
    THEN
        IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLS_exiting_bitmap[63] = 1
            THEN VM exit;
        FI;
FI;
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number)
```

```
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	NP	V/V	SGX1	This instruction is used to execute non-privileged Intel SGX leaf functions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.4

Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE← 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number
    THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
    THEN #GP(0); FI;

IN_64BIT_MODE ← IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and (CS.D = 0 or DS.B = 0)
```

```
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, or EACCEPTCOPY *)
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If operating in 16-bit mode.
	If data segment is in 16-bit mode.
	If CR0.PG = 0 or CR0.NE= 0.
#NM	If CR0.TS = 1.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.

	If CR0.NE= 0.
#NM	If CR0.TS = 1.

19. Updates to Chapter 1, Volume 4

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4:* Model Specific Registers.

Change to this chapter: Added reference to new volume 4: Model Specific Registers.

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*: *Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual’s content follows:

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

Chapter 2 — Model-Specific Registers (MSRs). Lists the MSRs available in the Pentium processors, the P6 family processors, the Pentium 4, Intel Xeon, Intel Core Solo, Intel Core Duo processors, and Intel Core 2 processor family and describes their functions.

1.3 NOTATIONAL CONVENTIONS

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

1.3.1 Bit and Byte Order

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

1.3.2 Reserved Bits and Software Compatibility

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

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

NOTE

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

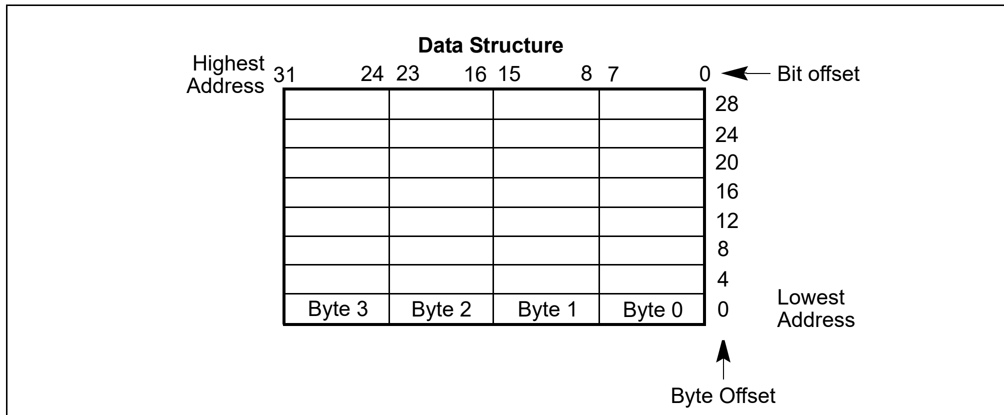


Figure 1-1. Bit and Byte Order

1.3.3 Instruction Operands

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

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

where:

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

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

For example:

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

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

1.3.4 Hexadecimal and Binary Numbers

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

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

1.3.5 Segmented Addressing

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

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

Segment-register:Byte-address

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

DS:FF79H

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

CS:EIP

1.3.6 Syntax for CPUID, CR, and MSR Values

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

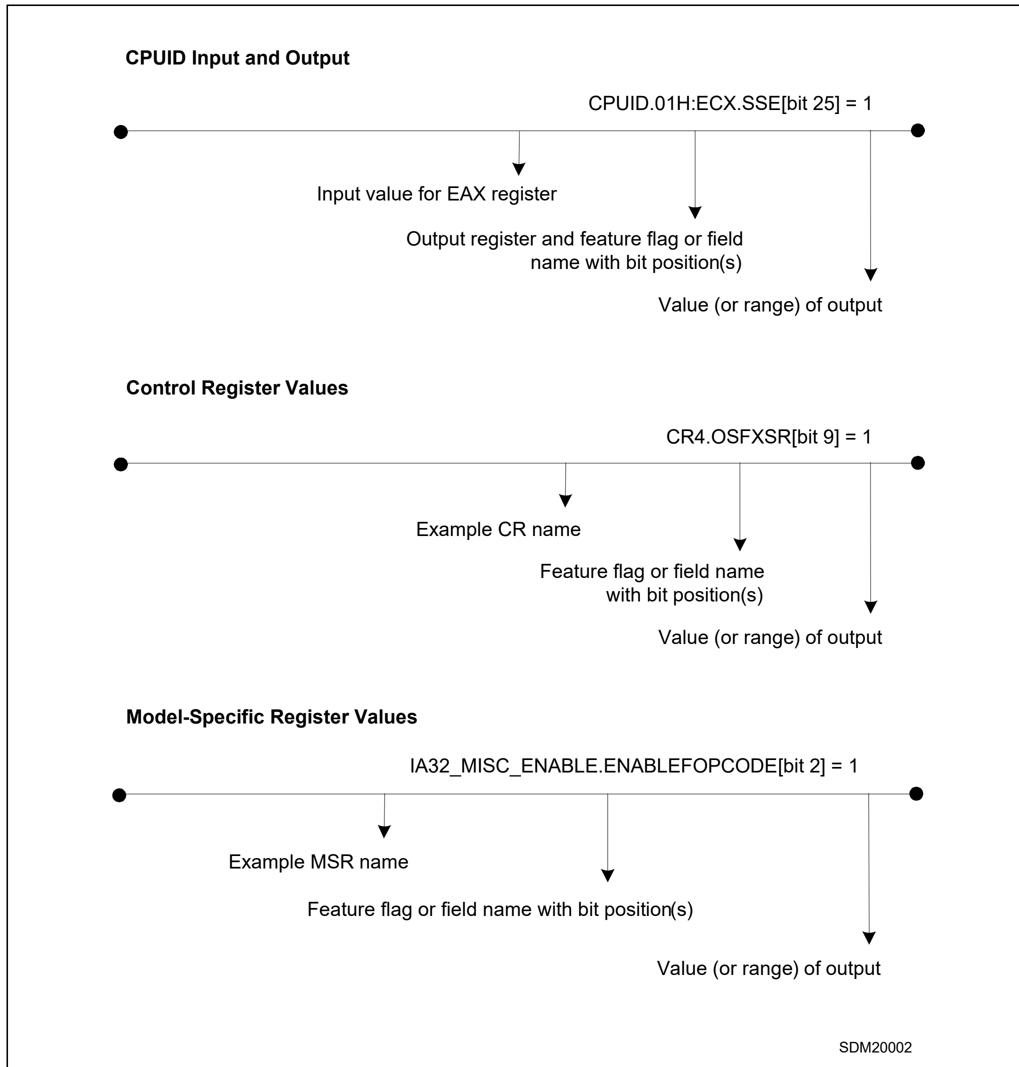


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.7 Exceptions

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

#PF(fault code)

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

#GP(0)

1.4 RELATED LITERATURE

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

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

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- Intel 64 Architecture x2APIC Specification:

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

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

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

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

More relevant links are:

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

20. Updates to Chapter 2, Volume 4

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4:* Model Specific Registers.

Changes to chapter: Added MSRs for processors based on Kaby Lake microarchitecture. Added MSR_POWER_CTL to Table 2-37 "Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture". Updated Table 2-40 "Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H".

CHAPTER 2
MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
06_85H	Future Intel® Xeon Phi™ Processor
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture
06_55H	Future Intel® Xeon® Processors
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Intel microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series
06_25H, 06_2CH	Intel Xeon processors 3600, 5600 series, Intel Core i7, i5 and i3 Processors

Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_1EH, 06_1FH	Intel Core i7 and i5 Processors
06_1AH	Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series
06_1DH	Intel Xeon processor MP 7400 series
06_17H	Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_5FH	Future Intel® Atom™ processors based on Goldmont Microarchitecture (code name Denverton)
06_5CH	Next Generation Intel Atom processors based on Goldmont Microarchitecture
06_4CH	Intel Atom processor X7-Z8000 and X5-Z8000 series based on Airmont Microarchitecture
06_5DH	Intel Atom processor X3-C3000 based on Silvermont Microarchitecture
06_5AH	Intel Atom processor Z3500 series
06_4AH	Intel Atom processor Z3400 series
06_37H	Intel Atom processor E3000 series, Z3600 series, Z3700 series
06_4DH	Intel Atom processor C2000 series
06_36H	Intel Atom processor S1000 Series
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	Intel Atom processor family, Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon processor, Intel Xeon processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily_DisplayModel = 05_09H and SteppingID = 0

2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these “architectural MSRs” were given the prefix “IA32_”. Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a machine specified MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF_DM" (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYADDR" in Table 2-2. "MAXPHYADDR" is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

Table 2-2. IA-32 Architectural MSRs

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 2.22, "MSRs in Pentium Processors."	**Pentium Processor (05_01H)**
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 2.22, "MSRs in Pentium Processors."	DF_DM = 05_01H
6H	6	IA32_MONITOR_FILTER_SIZE	See Section 8.10.5, "Monitor/Mwait Address Range Determination."	0F_03H
10H	16	IA32_TIME_STAMP_ COUNTER (TSC)	See Section 17.16, "Time-Stamp Counter."	05_01H
17H	23	IA32_PLATFORM_ID (MSR_PLATFORM_ID)	**Platform ID (RO)** The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.	06_01H
		49:0	Reserved.	
		52:50	**Platform Id (RO)** Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7	
		63:53	Reserved.	
1BH	27	IA32_APIC_BASE (APIC_BASE)		06_01H
		7:0	Reserved	
		8	BSP flag (R/W)	
		9	Reserved	
		10	Enable x2APIC mode	06_1AH
		11	APIC Global Enable (R/W)	
		(MAXPHYADDR - 1):12	APIC Base (R/W)	
		63: MAXPHYADDR	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
3AH	58	IA32_FEATURE_CONTROL	**Control Features in Intel 64 Processor (R/W)**	If any one enumeration condition for defined bit field holds
		0	Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written, writes to this bit will result in GP(0). Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.	If any one enumeration condition for defined bit field position greater than bit 0 holds
		1	Enable VMX inside SMX operation (R/WL): This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
		2	Enable VMX outside SMX operation (R/WL): This bit enables VMX for system executive that do not require SMX. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
		7:3	Reserved	
		14:8	SENTER Local Function Enables (R/WL): When set, each bit in the field represents an enable control for a corresponding SENTER function. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		15	SENTER Global Enable (R/WL): This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		16	Reserved	
		17	SGX Launch Control Enable (R/WL): This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
		18	SGX Global Enable (R/WL): This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		19	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		20	LMCE On (R/WL): When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
		63:21	Reserved	
3BH	59	IA32_TSC_ADJUST	Per Logical Processor TSC Adjust (R/Write to clear)	If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
		63:0	**THREAD_ADJUST:** Local offset value of the IA32_TSC for a logical processor. Reset value is Zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	
79H	121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 9.11.6, "Microcode Update Loader." A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
8BH	139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	BIOS Update Signature (RO) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
		31:0	Reserved	
		63:32	It is recommended that this field be pre-loaded with 0 prior to executing CPUID. If the field remains 0 following the execution of CPUID; this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature.	
8CH	140	IA32_SGXLEPUBKEYHASH0	IA32_SGXLEPUBKEYHASH[63:0] (R/W) Bits 63:0 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
8DH	141	IA32_SGXLEPUBKEYHASH1	IA32_SGXLEPUBKEYHASH[127:64] (R/W) Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8EH	142	IA32_SGXLEPUBKEYHASH2	IA32_SGXLEPUBKEYHASH[191:128] (R/W) Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8FH	143	IA32_SGXLEPUBKEYHASH3	IA32_SGXLEPUBKEYHASH[255:192] (R/W) Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	
9BH	155	IA32_SMM_MONITOR_CTL	SMM Monitor Configuration (R/W)	If CPUID.01H: ECX[5]=1 || CPUID.01H: ECX[6] = 1
		0	Valid (R/W)	
		1	Reserved	
		2	Controls SMI unblocking by VMXOFF (see Section 34.14.4)	If IA32_VMX_MISC[28]
		11:3	Reserved	
		31:12	MSEG Base (R/W)	
		63:32	Reserved	
9EH	158	IA32_SMBASE	Base address of the logical processor's SMRAM image (RO, SMM only)	If IA32_VMX_MISC[15]
C1H	193	IA32_PMC0 (PERFCTR0)	General Performance Counter 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
C2H	194	IA32_PMC1 (PERFCTR1)	General Performance Counter 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
C3H	195	IA32_PMC2	General Performance Counter 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
C4H	196	IA32_PMC3	General Performance Counter 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
C5H	197	IA32_PMC4	General Performance Counter 4 (R/W)	If CPUID.0AH: EAX[15:8] > 4
C6H	198	IA32_PMC5	General Performance Counter 5 (R/W)	If CPUID.0AH: EAX[15:8] > 5
C7H	199	IA32_PMC6	General Performance Counter 6 (R/W)	If CPUID.0AH: EAX[15:8] > 6

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	**C0_MCNT: C0 TSC Frequency Clock Count** Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_APERF.	
E8H	232	IA32_APERF	Actual Performance Clock Counter (R/Write to clear).	If CPUID.06H: ECX[0] = 1
		63:0	**C0_ACNT: C0 Actual Frequency Clock Count** Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	06_01H
		7:0	VCNT: The number of variable memory type ranges in the processor.	
		8	Fixed range MTRRs are supported when set.	
		9	Reserved.	
		10	WC Supported when set.	
		11	SMRR Supported when set.	
		63:12	Reserved.	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector	
		63:16	Reserved.	
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR (R/W)	06_01H
176H	374	IA32_SYSENTER_EIP	SYSENTER_EIP_MSR (R/W)	06_01H
179H	377	IA32_MCG_CAP (MCG_CAP)	Global Machine Check Capability (RO)	06_01H
		7:0	Count: Number of reporting banks.	
		8	MCG_CTL_P: IA32_MCG_CTL is present if this bit is set	
		9	MCG_EXT_P: Extended machine check state registers are present if this bit is set	
		10	MCP_CMCI_P: Support for corrected MC error event is present.	06_01H

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	MCG_TES_P: Threshold-based error status register are present if this bit is set.	
		15:12	Reserved	
		23:16	MCG_EXT_CNT: Number of extended machine check state registers present.	
		24	MCG_SER_P: The processor supports software error recovery if this bit is set.	
		25	Reserved.	
		26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.	06_3EH
		27	MCG_LMCE_P: Indicates that the processor support extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
		63:28	Reserved.	
17AH	378	IA32_MCG_STATUS (MCG_STATUS)	Global Machine Check Status (R/W0)	06_01H
		0	RIPV. Restart IP valid	06_01H
		1	EIPV. Error IP valid	06_01H
		2	MCIP. Machine check in progress	06_01H
		3	LMCE_S.	If IA32_MCG_CAP.LMCE_P[27] =1
		63:4	Reserved.	
17BH	379	IA32_MCG_CTL (MCG_CTL)	Global Machine Check Control (R/W)	If IA32_MCG_CAP.CTL_P[8] =1
180H-185H	384-389	Reserved		06_0EH[1]
186H	390	IA32_PERFEVTSEL0 (PERFEVTSEL0)	Performance Event Select Register 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
		7:0	Event Select: Selects a performance event logic unit.	
		15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	
		16	USR: Counts while in privilege level is not ring 0.	
		17	OS: Counts while in privilege level is ring 0.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		18	Edge: Enables edge detection if set.	
		19	PC: enables pin control.	
		20	INT: enables interrupt on counter overflow.	
		21	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
		22	EN: enables the corresponding performance counter to commence counting when this bit is set.	
		23	INV: invert the CMASK.	
		31:24	CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.	
		63:32	Reserved.	
187H	391	IA32_PERFEVTSEL1 (PERFEVTSEL1)	Performance Event Select Register 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
188H	392	IA32_PERFEVTSEL2	Performance Event Select Register 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
189H	393	IA32_PERFEVTSEL3	Performance Event Select Register 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
18AH-197H	394-407	Reserved		06_0EH[2]
198H	408	IA32_PERF_STATUS	(RO)	0F_03H
		15:0	Current performance State Value	
		63:16	Reserved.	
199H	409	IA32_PERF_CTL	(R/W)	0F_03H
		15:0	Target performance State Value	
		31:16	Reserved.	
		32	IDA Engage. (R/W) When set to 1: disengages IDA	06_0FH (Mobile only)
		63:33	Reserved.	
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.7.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1
		0	Extended On-Demand Clock Modulation Duty Cycle:	If CPUID.06H:EAX[5] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
		63:5	Reserved.	
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.7.2, "Thermal Monitor."	If CPUID.01H:EDX[22] = 1
		0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		7:5	Reserved.	
		14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
		15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
		23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved.	
19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.7.2, "Thermal Monitor"	If CPUID.01H:EDX[22] = 1
		0	Thermal Status (RO):	If CPUID.01H:EDX[22] = 1
		1	Thermal Status Log (R/W):	If CPUID.01H:EDX[22] = 1
		2	PROCHOT # or FORCEPR# event (RO)	If CPUID.01H:EDX[22] = 1
		3	PROCHOT # or FORCEPR# log (R/WC0)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WC0)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #2 log (R/WC0)	If CPUID.01H:ECX[8] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WC0)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved.	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1
		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved.	
1A0H	416	IA32_MISC_ENABLE	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.	
		0	**Fast-Strings Enable** When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default); when clear, fast-strings are disabled.	0F_0H
		2:1	Reserved.	
		3	**Automatic Thermal Control Circuit Enable (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated. The default value of this field varies with product . See respective tables where default value is listed.	0F_0H
		6:4	Reserved	
		7	**Performance Monitoring Available (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled	0F_0H
		10:8	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	**Branch Trace Storage Unavailable (RO)** 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported	0F_0H
		12	**Processor Event Based Sampling (PEBS) Unavailable (RO)** 1 = PEBS is not supported; 0 = PEBS is supported.	06_0FH
		15:13	Reserved.	
		16	**Enhanced Intel SpeedStep Technology Enable (R/W)** 0= Enhanced Intel SpeedStep Technology disabled 1 = Enhanced Intel SpeedStep Technology enabled	If CPUID.01H: ECX[7] =1
		17	Reserved.	
		18	**ENABLE MONITOR FSM (R/W)** When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
		21:19	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		22	**Limit CPUID Maxval (R/W)** When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2. BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2. Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported. Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception. Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.	0F_03H
		23	**xTPR Message Disable (R/W)** When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.	if CPUID.01H:ECX[14] = 1
		33:24	Reserved.	
		34	**XD Bit Disable (R/W)** When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location, if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.	if CPUID.80000001H:EDX[20] = 1
		63:35	Reserved.	
1B0H	432	IA32_ENERGY_PERF_BIAS	Performance Energy Bias Hint (R/W)	if CPUID.6H:ECX[3] = 1
		3:0	Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.	
		63:4	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg Thermal Status (RO):	
		1	Pkg Thermal Status Log (R/W):	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WC0)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status log (R/WC0)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 log (R/WC0)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 log (R/WC0)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation log (R/WC0)	
		15:12	Reserved.	
		22:16	Pkg Digital Readout (RO)	
		63:23	Reserved.	
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved.	
		4	Pkg Overheat Interrupt Enable	
		7:5	Reserved.	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
		22:16	Pkg Threshold #2 Value	
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address Hex	Register Address Decimal	Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
		63:25	Reserved.	
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved.	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
		15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
		63:16	Reserved.	
1F2H	498	IA32_SMRR_PHYSBASE	**SMRR Base Address (Writeable only in SMM)** Base address of SMM memory range.	If IA32_MTRRCAP.SMRR[11] = 1
		7:0	Type. Specifies memory type of the range.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address Hex	Register Address Decimal	Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
		11:8	Reserved.	
		31:12	**PhysBase.** SMRR physical Base Address.	
		63:32	Reserved.	
1F3H	499	IA32_SMRR_PHYSMASK	**SMRR Range Mask. (Writeable only in SMM)** Range Mask of SMM memory range.	If IA32_MTRRCAP[SMRR] = 1
		10:0	Reserved.	
		11	**Valid** Enable range mask.	
		31:12	**PhysMask** SMRR address range mask.	
		63:32	Reserved.	
1F8H	504	IA32_PLATFORM_DCA_CAP	DCA Capability (R)	If CPUID.01H: ECX[18] = 1
1F9H	505	IA32_CPU_DCA_CAP	If set, CPU supports Prefetch-Hint type.	If CPUID.01H: ECX[18] = 1
1FAH	506	IA32_DCA_0_CAP	DCA type 0 Status and Control register.	If CPUID.01H: ECX[18] = 1
		0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
		2:1	TRANSACTION	
		6:3	DCA_TYPE	
		10:7	DCA_QUEUE_SIZE	
		12:11	Reserved.	
		16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
		23:17	Reserved.	
		24	SW_BLOCK: SW can request DCA block by setting this bit.	
		25	Reserved.	
		26	HW_BLOCK: Set when DCA is blocked by HW (e.g. CR0.CD = 1).	
		31:27	Reserved.	
200H	512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	See Section 11.11.2.3, “Variable Range MTRRs.”	If CPUID.01H: EDX.MTRR[12] =1
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If CPUID.01H: EDX.MTRR[12] =1
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If CPUID.01H: EDX.MTRR[12] =1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If CPUID.01H: EDX.MTRR[12] =1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If CPUID.01H: EDX.MTRR[12] =1
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If CPUID.01H: EDX.MTRR[12] =1
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If CPUID.01H: EDX.MTRR[12] =1
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If CPUID.01H: EDX.MTRR[12] =1
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If CPUID.01H: EDX.MTRR[12] =1
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If CPUID.01H: EDX.MTRR[12] =1
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If CPUID.01H: EDX.MTRR[12] =1
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If CPUID.01H: EDX.MTRR[12] =1
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If CPUID.01H: EDX.MTRR[12] =1
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If CPUID.01H: EDX.MTRR[12] =1
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If CPUID.01H: EDX.MTRR[12] =1
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If CPUID.01H: EDX.MTRR[12] =1
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	if IA32_MTRRCAP[7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	if IA32_MTRRCAP[7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	if IA32_MTRRCAP[7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	if IA32_MTRRCAP[7:0] > 9
250H	592	IA32_MTRR_FIX64K_00000	MTRRfix64K_00000	If CPUID.01H: EDX.MTRR[12] =1
258H	600	IA32_MTRR_FIX16K_80000	MTRRfix16K_80000	If CPUID.01H: EDX.MTRR[12] =1
259H	601	IA32_MTRR_FIX16K_A0000	MTRRfix16K_A0000	If CPUID.01H: EDX.MTRR[12] =1
268H	616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)	See Section 11.11.2.2, “Fixed Range MTRRs.”	If CPUID.01H: EDX.MTRR[12] =1
269H	617	IA32_MTRR_FIX4K_C8000	MTRRfix4K_C8000	If CPUID.01H: EDX.MTRR[12] =1
26AH	618	IA32_MTRR_FIX4K_D0000	MTRRfix4K_D0000	If CPUID.01H: EDX.MTRR[12] =1
26BH	619	IA32_MTRR_FIX4K_D8000	MTRRfix4K_D8000	If CPUID.01H: EDX.MTRR[12] =1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
26CH	620	IA32_MTRR_FIX4K_E0000	MTRRfix4K_E0000	If CPUID.01H: EDX.MTRR[12] =1
26DH	621	IA32_MTRR_FIX4K_E8000	MTRRfix4K_E8000	If CPUID.01H: EDX.MTRR[12] =1
26EH	622	IA32_MTRR_FIX4K_F0000	MTRRfix4K_F0000	If CPUID.01H: EDX.MTRR[12] =1
26FH	623	IA32_MTRR_FIX4K_F8000	MTRRfix4K_F8000	If CPUID.01H: EDX.MTRR[12] =1
277H	631	IA32_PAT	IA32_PAT (R/W)	If CPUID.01H: EDX.MTRR[16] =1
		2:0	PA0	
		7:3	Reserved.	
		10:8	PA1	
		15:11	Reserved.	
		18:16	PA2	
		23:19	Reserved.	
		26:24	PA3	
		31:27	Reserved.	
		34:32	PA4	
		39:35	Reserved.	
		42:40	PA5	
		47:43	Reserved.	
		50:48	PA6	
		55:51	Reserved.	
		58:56	PA7	
		63:59	Reserved.	
280H	640	IA32_MC0_CTL2	(R/W)	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 0
		14:0	Corrected error count threshold.	
		29:15	Reserved.	
		30	CMCI_EN	
		63:31	Reserved.	
281H	641	IA32_MC1_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
283H	643	IA32_MC3_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6
287H	647	IA32_MC7_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
28CH	652	IA32_MC12_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
292H	658	IA32_MC18_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21
296H	662	IA32_MC22_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
29BH	667	IA32_MC27_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
2FFH	767	IA32_MTRR_DEF_TYPE	MTRRdefType (R/W)	If CPUID.01H: EDX.MTRR[12] =1
		2:0	Default Memory Type	
		9:3	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved.	
309H	777	IA32_FIXED_CTR0 (MSR_PERF_FIXED_CTR0)	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1 (MSR_PERF_FIXED_CTR1)	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2 (MSR_PERF_FIXED_CTR2)	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref	If CPUID.0AH: EDX[4:0] > 2
345H	837	IA32_PERF_CAPABILITIES	RO	If CPUID.01H: ECX[15] = 1
		5:0	LBR format	
		6	PEBS Trap	
		7	PEBSSaveArchRegs	
		11:8	PEBS Record Format	
		12	1: Freeze while SMM is supported.	
		13	1: Full width of counter writable via IA32_A_PMCx.	
		63:14	Reserved.	
38DH	909	IA32_FIXED_CTR_CTRL	Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0.	
		1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
		2	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		3	EN0_PMI: Enable PMI when fixed counter 0 overflows.	
		4	EN1_OS: Enable Fixed Counter 1to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1to count while CPL > 0.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		6	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
		10	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
		63:12	Reserved.	
38EH	910	IA32_PERF_GLOBAL_STATUS	Global Performance Counter Status (RO)	If CPUID.0AH: EAX[7:0] > 0
		0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
		1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
		2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
		3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3
		31:4	Reserved.	
		32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
		33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1
		34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
		54:35	Reserved.	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer was completely filled.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		58	LBR_Frz: LBRs are frozen due to ▪ IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1, ▪ The LBR stack overflowed	If CPUID.0AH: EAX[7:0] > 3
		59	CTR_Frz: Performance counters in the core PMU are frozen due to ▪ IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1, ▪ one or more core PMU counters overflowed.	If CPUID.0AH: EAX[7:0] > 3
		60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation intel SGX to protect an enclave.	If CPUID.(EAX=07H, ECX=0):EBX[2] = 1
		61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.0AH: EAX[7:0] > 2
		62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.0AH: EAX[7:0] > 0
		63	CondChgd: status bits of this register has changed.	If CPUID.0AH: EAX[7:0] > 0
38FH	911	IA32_PERF_GLOBAL_CTRL	Global Performance Counter Control (R/W) Counter increments while the result of ANDing respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.	If CPUID.0AH: EAX[7:0] > 0
		0	EN_PMC0	If CPUID.0AH: EAX[15:8] > 0
		1	EN_PMC1	If CPUID.0AH: EAX[15:8] > 1
		2	EN_PMC2	If CPUID.0AH: EAX[15:8] > 2
		n	EN_PMCn	If CPUID.0AH: EAX[15:8] > n
		31:n+1	Reserved.	
		32	EN_FIXED_CTR0	If CPUID.0AH: EDX[4:0] > 0
		33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2
		63:35	Reserved.	
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Global Performance Counter Overflow Control (R/W)	If CPUID.0AH: EAX[7:0] > 0 && CPUID.0AH: EAX[7:0] <= 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		60:56	Reserved.	
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Global Performance Counter Overflow Reset Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA[8] = 1
		57:56	Reserved.	
		58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to Clear ASCI bit.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Global Performance Counter Overflow Set Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		1	Set 1 to cause Ovf_PMC1 = 1	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to cause Ovf_PMC2 = 1	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to cause Ovf_PMCn = 1	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.0AH: EAX[7:0] > 3
		34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.0AH: EAX[7:0] > 3
		54:35	Reserved.	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.0AH: EAX[7:0] > 3
		57:56	Reserved.	
		58	Set 1 to cause LBR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to cause CTR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to cause ASCI = 1.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.0AH: EAX[7:0] > 3
		62	Set 1 to cause OvfBuf = 1.	If CPUID.0AH: EAX[7:0] > 3
		63	Reserved	
392H	914	IA32_PERF_GLOBAL_INUSE	Indicator of core perfmon interface is in use (RO)	If CPUID.0AH: EAX[7:0] > 3
		0	IA32_PERFEVTSEL0 in use	
		1	IA32_PERFEVTSEL1 in use	If CPUID.0AH: EAX[15:8] > 1
		2	IA32_PERFEVTSEL2 in use	If CPUID.0AH: EAX[15:8] > 2
		n	IA32_PERFEVTSELn in use	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	IA32_FIXED_CTR0 in use	
		33	IA32_FIXED_CTR1 in use	
		34	IA32_FIXED_CTR2 in use	
		62:35	Reserved or Model specific.	
		63	PMI in use.	
3F1H	1009	IA32_PEBS_ENABLE	PEBS Control (R/W)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	Enable PEBS on IA32_PMC0.	06_0FH
		3:1	Reserved or Model specific.	
		31:4	Reserved.	
		35:32	Reserved or Model specific.	
		63:36	Reserved.	
400H	1024	IA32_MC0_CTL	MC0_CTL	If IA32_MCG_CAP.CNT >0
401H	1025	IA32_MC0_STATUS	MC0_STATUS	If IA32_MCG_CAP.CNT >0
402H	1026	IA32_MC0_ADDR [1]	MC0_ADDR	If IA32_MCG_CAP.CNT >0
403H	1027	IA32_MC0_MISC	MC0_MISC	If IA32_MCG_CAP.CNT >0
404H	1028	IA32_MC1_CTL	MC1_CTL	If IA32_MCG_CAP.CNT >1
405H	1029	IA32_MC1_STATUS	MC1_STATUS	If IA32_MCG_CAP.CNT >1
406H	1030	IA32_MC1_ADDR [2]	MC1_ADDR	If IA32_MCG_CAP.CNT >1
407H	1031	IA32_MC1_MISC	MC1_MISC	If IA32_MCG_CAP.CNT >1
408H	1032	IA32_MC2_CTL	MC2_CTL	If IA32_MCG_CAP.CNT >2
409H	1033	IA32_MC2_STATUS	MC2_STATUS	If IA32_MCG_CAP.CNT >2
40AH	1034	IA32_MC2_ADDR [1]	MC2_ADDR	If IA32_MCG_CAP.CNT >2
40BH	1035	IA32_MC2_MISC	MC2_MISC	If IA32_MCG_CAP.CNT >2
40CH	1036	IA32_MC3_CTL	MC3_CTL	If IA32_MCG_CAP.CNT >3
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	If IA32_MCG_CAP.CNT >3
40EH	1038	IA32_MC3_ADDR [1]	MC3_ADDR	If IA32_MCG_CAP.CNT >3
40FH	1039	IA32_MC3_MISC	MC3_MISC	If IA32_MCG_CAP.CNT >3
410H	1040	IA32_MC4_CTL	MC4_CTL	If IA32_MCG_CAP.CNT >4
411H	1041	IA32_MC4_STATUS	MC4_STATUS	If IA32_MCG_CAP.CNT >4
412H	1042	IA32_MC4_ADDR [1]	MC4_ADDR	If IA32_MCG_CAP.CNT >4
413H	1043	IA32_MC4_MISC	MC4_MISC	If IA32_MCG_CAP.CNT >4
414H	1044	IA32_MC5_CTL	MC5_CTL	If IA32_MCG_CAP.CNT >5
415H	1045	IA32_MC5_STATUS	MC5_STATUS	If IA32_MCG_CAP.CNT >5
416H	1046	IA32_MC5_ADDR [1]	MC5_ADDR	If IA32_MCG_CAP.CNT >5
417H	1047	IA32_MC5_MISC	MC5_MISC	If IA32_MCG_CAP.CNT >5
418H	1048	IA32_MC6_CTL	MC6_CTL	If IA32_MCG_CAP.CNT >6
419H	1049	IA32_MC6_STATUS	MC6_STATUS	If IA32_MCG_CAP.CNT >6
41AH	1050	IA32_MC6_ADDR [1]	MC6_ADDR	If IA32_MCG_CAP.CNT >6
41BH	1051	IA32_MC6_MISC	MC6_MISC	If IA32_MCG_CAP.CNT >6
41CH	1052	IA32_MC7_CTL	MC7_CTL	If IA32_MCG_CAP.CNT >7
41DH	1053	IA32_MC7_STATUS	MC7_STATUS	If IA32_MCG_CAP.CNT >7
41EH	1054	IA32_MC7_ADDR [1]	MC7_ADDR	If IA32_MCG_CAP.CNT >7
41FH	1055	IA32_MC7_MISC	MC7_MISC	If IA32_MCG_CAP.CNT >7

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
420H	1056	IA32_MC8_CTL	MC8_CTL	If IA32_MCG_CAP.CNT >8
421H	1057	IA32_MC8_STATUS	MC8_STATUS	If IA32_MCG_CAP.CNT >8
422H	1058	IA32_MC8_ADDR[1]	MC8_ADDR	If IA32_MCG_CAP.CNT >8
423H	1059	IA32_MC8_MISC	MC8_MISC	If IA32_MCG_CAP.CNT >8
424H	1060	IA32_MC9_CTL	MC9_CTL	If IA32_MCG_CAP.CNT >9
425H	1061	IA32_MC9_STATUS	MC9_STATUS	If IA32_MCG_CAP.CNT >9
426H	1062	IA32_MC9_ADDR[1]	MC9_ADDR	If IA32_MCG_CAP.CNT >9
427H	1063	IA32_MC9_MISC	MC9_MISC	If IA32_MCG_CAP.CNT >9
428H	1064	IA32_MC10_CTL	MC10_CTL	If IA32_MCG_CAP.CNT >10
429H	1065	IA32_MC10_STATUS	MC10_STATUS	If IA32_MCG_CAP.CNT >10
42AH	1066	IA32_MC10_ADDR[1]	MC10_ADDR	If IA32_MCG_CAP.CNT >10
42BH	1067	IA32_MC10_MISC	MC10_MISC	If IA32_MCG_CAP.CNT >10
42CH	1068	IA32_MC11_CTL	MC11_CTL	If IA32_MCG_CAP.CNT >11
42DH	1069	IA32_MC11_STATUS	MC11_STATUS	If IA32_MCG_CAP.CNT >11
42EH	1070	IA32_MC11_ADDR[1]	MC11_ADDR	If IA32_MCG_CAP.CNT >11
42FH	1071	IA32_MC11_MISC	MC11_MISC	If IA32_MCG_CAP.CNT >11
430H	1072	IA32_MC12_CTL	MC12_CTL	If IA32_MCG_CAP.CNT >12
431H	1073	IA32_MC12_STATUS	MC12_STATUS	If IA32_MCG_CAP.CNT >12
432H	1074	IA32_MC12_ADDR[1]	MC12_ADDR	If IA32_MCG_CAP.CNT >12
433H	1075	IA32_MC12_MISC	MC12_MISC	If IA32_MCG_CAP.CNT >12
434H	1076	IA32_MC13_CTL	MC13_CTL	If IA32_MCG_CAP.CNT >13
435H	1077	IA32_MC13_STATUS	MC13_STATUS	If IA32_MCG_CAP.CNT >13
436H	1078	IA32_MC13_ADDR[1]	MC13_ADDR	If IA32_MCG_CAP.CNT >13
437H	1079	IA32_MC13_MISC	MC13_MISC	If IA32_MCG_CAP.CNT >13
438H	1080	IA32_MC14_CTL	MC14_CTL	If IA32_MCG_CAP.CNT >14
439H	1081	IA32_MC14_STATUS	MC14_STATUS	If IA32_MCG_CAP.CNT >14
43AH	1082	IA32_MC14_ADDR[1]	MC14_ADDR	If IA32_MCG_CAP.CNT >14
43BH	1083	IA32_MC14_MISC	MC14_MISC	If IA32_MCG_CAP.CNT >14
43CH	1084	IA32_MC15_CTL	MC15_CTL	If IA32_MCG_CAP.CNT >15
43DH	1085	IA32_MC15_STATUS	MC15_STATUS	If IA32_MCG_CAP.CNT >15
43EH	1086	IA32_MC15_ADDR[1]	MC15_ADDR	If IA32_MCG_CAP.CNT >15
43FH	1087	IA32_MC15_MISC	MC15_MISC	If IA32_MCG_CAP.CNT >15
440H	1088	IA32_MC16_CTL	MC16_CTL	If IA32_MCG_CAP.CNT >16
441H	1089	IA32_MC16_STATUS	MC16_STATUS	If IA32_MCG_CAP.CNT >16
442H	1090	IA32_MC16_ADDR[1]	MC16_ADDR	If IA32_MCG_CAP.CNT >16
443H	1091	IA32_MC16_MISC	MC16_MISC	If IA32_MCG_CAP.CNT >16
444H	1092	IA32_MC17_CTL	MC17_CTL	If IA32_MCG_CAP.CNT >17

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
445H	1093	IA32_MC17_STATUS	MC17_STATUS	If IA32_MCG_CAP.CNT >17
446H	1094	IA32_MC17_ADDR[1]	MC17_ADDR	If IA32_MCG_CAP.CNT >17
447H	1095	IA32_MC17_MISC	MC17_MISC	If IA32_MCG_CAP.CNT >17
448H	1096	IA32_MC18_CTL	MC18_CTL	If IA32_MCG_CAP.CNT >18
449H	1097	IA32_MC18_STATUS	MC18_STATUS	If IA32_MCG_CAP.CNT >18
44AH	1098	IA32_MC18_ADDR[1]	MC18_ADDR	If IA32_MCG_CAP.CNT >18
44BH	1099	IA32_MC18_MISC	MC18_MISC	If IA32_MCG_CAP.CNT >18
44CH	1100	IA32_MC19_CTL	MC19_CTL	If IA32_MCG_CAP.CNT >19
44DH	1101	IA32_MC19_STATUS	MC19_STATUS	If IA32_MCG_CAP.CNT >19
44EH	1102	IA32_MC19_ADDR[1]	MC19_ADDR	If IA32_MCG_CAP.CNT >19
44FH	1103	IA32_MC19_MISC	MC19_MISC	If IA32_MCG_CAP.CNT >19
450H	1104	IA32_MC20_CTL	MC20_CTL	If IA32_MCG_CAP.CNT >20
451H	1105	IA32_MC20_STATUS	MC20_STATUS	If IA32_MCG_CAP.CNT >20
452H	1106	IA32_MC20_ADDR[1]	MC20_ADDR	If IA32_MCG_CAP.CNT >20
453H	1107	IA32_MC20_MISC	MC20_MISC	If IA32_MCG_CAP.CNT >20
454H	1108	IA32_MC21_CTL	MC21_CTL	If IA32_MCG_CAP.CNT >21
455H	1109	IA32_MC21_STATUS	MC21_STATUS	If IA32_MCG_CAP.CNT >21
456H	1110	IA32_MC21_ADDR[1]	MC21_ADDR	If IA32_MCG_CAP.CNT >21
457H	1111	IA32_MC21_MISC	MC21_MISC	If IA32_MCG_CAP.CNT >21
458H		IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H		IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH		IA32_MC22_ADDR[1]	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH		IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH		IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH		IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH		IA32_MC23_ADDR[1]	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH		IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H		IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H		IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H		IA32_MC24_ADDR[1]	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H		IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H		IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H		IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H		IA32_MC25_ADDR[1]	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H		IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H		IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H		IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
46AH		IA32_MC26_ADDR[1]	MC26_ADDR	If IA32_MCG_CAP.CNT >26
46BH		IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT >26
46CH		IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT >27
46DH		IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT >27
46EH		IA32_MC27_ADDR[1]	MC27_ADDR	If IA32_MCG_CAP.CNT >27
46FH		IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT >27
470H		IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT >28
471H		IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT >28
472H		IA32_MC28_ADDR[1]	MC28_ADDR	If IA32_MCG_CAP.CNT >28
473H		IA32_MC28_MISC	MC28_MISC	If IA32_MCG_CAP.CNT >28
480H	1152	IA32_VMX_BASIC	**Reporting Register of Basic VMX Capabilities (R/O)** See Appendix A.1, “Basic VMX Information.”	If CPUID.01H:ECX.[5] = 1
481H	1153	IA32_VMX_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3.1, “Pin-Based VM-Execution Controls.”	If CPUID.01H:ECX.[5] = 1
482H	1154	IA32_VMX_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3.2, “Primary Processor-Based VM-Execution Controls.”	If CPUID.01H:ECX.[5] = 1
483H	1155	IA32_VMX_EXIT_CTLS	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, “VM-Exit Controls.”	If CPUID.01H:ECX.[5] = 1
484H	1156	IA32_VMX_ENTRY_CTLS	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, “VM-Entry Controls.”	If CPUID.01H:ECX.[5] = 1
485H	1157	IA32_VMX_MISC	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, “Miscellaneous Data.”	If CPUID.01H:ECX.[5] = 1
486H	1158	IA32_VMX_CR0_FIXED0	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, “VMX-Fixed Bits in CR0.”	If CPUID.01H:ECX.[5] = 1
487H	1159	IA32_VMX_CR0_FIXED1	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, “VMX-Fixed Bits in CR0.”	If CPUID.01H:ECX.[5] = 1
488H	1160	IA32_VMX_CR4_FIXED0	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, “VMX-Fixed Bits in CR4.”	If CPUID.01H:ECX.[5] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
489H	1161	IA32_VMX_CR4_FIXED1	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
48AH	1162	IA32_VMX_VMCS_ENUM	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration."	If CPUID.01H:ECX.[5] = 1
48BH	1163	IA32_VMX_PROCBASED_CTLS2	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3.3, "Secondary Processor-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63])
48CH	1164	IA32_VMX_EPT_VPID_CAP	**Capability Reporting Register of EPT and VPID (R/O)** See Appendix A.10, "VPID and EPT Capabilities."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63] && (IA32_VMX_PROCBASED_CTLS2[33] || IA32_VMX_PROCBASED_CTLS2[37]))
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Appendix A.4, "VM-Exit Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Appendix A.5, "VM-Entry Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
491H	1169	IA32_VMX_VMFUNC	**Capability Reporting Register of VM-function Controls (R/O)**	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1
4D0H	1232	IA32_MCG_EXT_CTL	(R/W)	If IA32_MCG_CAP.LMCE_P =1
		0	LMCE_EN.	
		63:1	Reserved.	
500H	1280	IA32_SGX_SVN_STATUS	Status and SVN Threshold of SGX Support for ACM (RO).	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		0	Lock.	See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1	Reserved.	
		23:16	SGX_SVN_SINIT.	See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
560H	1376	IA32_RTIT_OUTPUT_BASE	**Trace Output Base Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		MAXPHYADDR[3]-1:7	Base physical address	
		63:MAXPHYADDR	**Reserved.**	
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	**Trace Output Mask Pointers Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	**Output Offset.**	
570H	1392	IA32_RTIT_CTL	**Trace Control Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**TraceEn**	
		1	**CYCEn**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		2	**OS**	
		3	**User**	
		5:4	Reserved,	
		6	**FabricEn**	If (CPUID.(EAX=07H, ECX=0):ECX[3] = 1)
		7	**CR3 filter**	
		8	**ToPA**	
		9	**MTCEn**	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		10	**TSCEn**	
		11	**DisRETC**	
		12	Reserved, MBZ	
		13	**BranchEn**	
		17:14	**MTCFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		18	Reserved, MBZ	
		22:19	**CYCThresh**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		23	Reserved, MBZ	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		27:24	**PSBFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		31:28	Reserved, MBZ	
		35:32	**ADDR0_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		39:36	**ADDR1_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		43:40	**ADDR2_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:44	**ADDR3_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		63:48	Reserved, MBZ.	
571H	1393	IA32_RTIT_STATUS	**Tracing Status Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**FilterEn** (writes ignored)	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)
		1	**ContexEn** (writes ignored)	
		2	**TriggerEn** (writes ignored)	
		3	Reserved	
		4	**Error**	
		5	**Stopped**	
		31:6	Reserved, MBZ	
		48:32	**PacketByteCnt**	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
		63:49	**Reserved**	
572H	1394	IA32_RTIT_CR3_MATCH	**Trace Filter CR3 Match Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match	
580H	1408	IA32_RTIT_ADDR0_A	**Region 0 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
581H	1409	IA32_RTIT_ADDR0_B	**Region 0 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
582H	1410	IA32_RTIT_ADDR1_A	**Region 1 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
583H	1411	IA32_RTIT_ADDR1_B	**Region 1 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	
584H	1412	IA32_RTIT_ADDR2_A	**Region 2 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
585H	1413	IA32_RTIT_ADDR2_B	**Region 2 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
586H	1414	IA32_RTIT_ADDR3_A	**Region 3 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
587H	1415	IA32_RTIT_ADDR3_B	**Region 3 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
600H	1536	IA32_DS_AREA	**DS Save Area (R/W)** Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.15.4, "Debug Store (DS) Mechanism."	If(CPUID.01H:EDX.DS[21] = 1
		63:0	The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.	
		31:0	The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.	
		63:32	Reserved if not in IA-32e mode.	
6E0H	1760	IA32_TSC_DEADLINE	**TSC Target of Local APIC's TSC Deadline Mode (R/W)**	If CPUID.01H:ECX.[24] = 1
770H	1904	IA32_PM_ENABLE	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[7] = 1
		0	**HWP_ENABLE (R/W1-Once)** See Section 14.4.2, "Enabling HWP"	If CPUID.06H:EAX.[7] = 1
		63:1	Reserved.	
771H	1905	IA32_HWP_CAPABILITIES	**HWP Performance Range Enumeration (RO)**	If CPUID.06H:EAX.[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:0	**Highest_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		15:8	**Guaranteed_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		23:16	**Most_Efficient_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		31:24	**Lowest_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		63:32	Reserved.	
772H	1906	IA32_HWP_REQUEST_PKG	**Power Management Control Hints for All Logical Processors in a Package (R/W)**	If CPUID.06H:EAX.[11] = 1
		7:0	**Minimum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		15:8	**Maximum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		23:16	**Desired_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		31:24	**Energy_Performance_Preference** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	**Activity_Window** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[9] = 1
		63:42	Reserved.	
773H	1907	IA32_HWP_INTERRUPT	**Control HWP Native Interrupts (R/W)**	If CPUID.06H:EAX.[8] = 1
		0	**EN_Guaranteed_Performance_Change** See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		1	**EN_Excursion_Minimum** See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		63:2	Reserved.	
774H	1908	IA32_HWP_REQUEST	**Power Management Control Hints to a Logical Processor (R/W)**	If CPUID.06H:EAX.[7] = 1
		7:0	**Minimum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		15:8	**Maximum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		23:16	**Desired_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		31:24	**Energy_Performance_Preference** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	**Activity_Window** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[9] = 1
		42	**Package_Control** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[11] = 1
		63:43	Reserved.	
777H	1911	IA32_HWP_STATUS	**Log bits indicating changes to Guaranteed & excursions to Minimum (R/W)**	If CPUID.06H:EAX.[7] = 1
		0	**Guaranteed_Performance_Change (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		1	Reserved.	
		2	**Excursion_To_Minimum (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved.	
802H	2050	IA32_X2APIC_APICID	**x2APIC ID Register (R/O)** See x2APIC Specification	If CPUID.01H:ECX[21] = 1 && IA32_APIC_BASE.[10] = 1
803H	2051	IA32_X2APIC_VERSION	**x2APIC Version Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
808H	2056	IA32_X2APIC_TPR	**x2APIC Task Priority Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80AH	2058	IA32_X2APIC_PPR	**x2APIC Processor Priority Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80BH	2059	IA32_X2APIC_EOI	**x2APIC EOI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80DH	2061	IA32_X2APIC_LDR	**x2APIC Logical Destination Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80FH	2063	IA32_X2APIC_SIVR	**x2APIC Spurious Interrupt Vector Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
810H	2064	IA32_X2APIC_ISR0	**x2APIC In-Service Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
811H	2065	IA32_X2APIC_ISR1	**x2APIC In-Service Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
812H	2066	IA32_X2APIC_ISR2	**x2APIC In-Service Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
813H	2067	IA32_X2APIC_ISR3	**x2APIC In-Service Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
814H	2068	IA32_X2APIC_ISR4	**x2APIC In-Service Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
815H	2069	IA32_X2APIC_ISR5	**x2APIC In-Service Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
816H	2070	IA32_X2APIC_ISR6	**x2APIC In-Service Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
817H	2071	IA32_X2APIC_ISR7	**x2APIC In-Service Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
818H	2072	IA32_X2APIC_TMR0	**x2APIC Trigger Mode Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
819H	2073	IA32_X2APIC_TMR1	**x2APIC Trigger Mode Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81AH	2074	IA32_X2APIC_TMR2	**x2APIC Trigger Mode Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81BH	2075	IA32_X2APIC_TMR3	**x2APIC Trigger Mode Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81CH	2076	IA32_X2APIC_TMR4	**x2APIC Trigger Mode Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81DH	2077	IA32_X2APIC_TMR5	**x2APIC Trigger Mode Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81EH	2078	IA32_X2APIC_TMR6	**x2APIC Trigger Mode Register Bits 223:192 (R/O)**	If (CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
81FH	2079	IA32_X2APIC_TMR7	**x2APIC Trigger Mode Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
820H	2080	IA32_X2APIC_IRR0	**x2APIC Interrupt Request Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
821H	2081	IA32_X2APIC_IRR1	**x2APIC Interrupt Request Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
822H	2082	IA32_X2APIC_IRR2	**x2APIC Interrupt Request Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
823H	2083	IA32_X2APIC_IRR3	**x2APIC Interrupt Request Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
824H	2084	IA32_X2APIC_IRR4	**x2APIC Interrupt Request Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
825H	2085	IA32_X2APIC_IRR5	**x2APIC Interrupt Request Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
826H	2086	IA32_X2APIC_IRR6	**x2APIC Interrupt Request Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
827H	2087	IA32_X2APIC_IRR7	**x2APIC Interrupt Request Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
828H	2088	IA32_X2APIC_ESR	**x2APIC Error Status Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
82FH	2095	IA32_X2APIC_LVT_CMCI	**x2APIC LVT Corrected Machine Check Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
830H	2096	IA32_X2APIC_ICR	**x2APIC Interrupt Command Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
832H	2098	IA32_X2APIC_LVT_TIMER	**x2APIC LVT Timer Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
833H	2099	IA32_X2APIC_LVT_THERMAL	**x2APIC LVT Thermal Sensor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
834H	2100	IA32_X2APIC_LVT_PMI	**x2APIC LVT Performance Monitor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address Hex	Register Address Decimal	Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
835H	2101	IA32_X2APIC_LVT_LINT0	**x2APIC LVT LINT0 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
836H	2102	IA32_X2APIC_LVT_LINT1	**x2APIC LVT LINT1 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
837H	2103	IA32_X2APIC_LVT_ERROR	**x2APIC LVT Error Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
838H	2104	IA32_X2APIC_INIT_COUNT	**x2APIC Initial Count Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
839H	2105	IA32_X2APIC_CUR_COUNT	**x2APIC Current Count Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83EH	2110	IA32_X2APIC_DIV_CONF	**x2APIC Divide Configuration Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83FH	2111	IA32_X2APIC_SELF_IPI	**x2APIC Self IPI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
C80H	3200	IA32_DEBUG_INTERFACE	**Silicon Debug Feature Control (R/W)**	If CPUID.01H:ECX.[11] = 1
		0	**Enable (R/W)** BIOS set 1 to enable Silicon debug features. Default is 0	If CPUID.01H:ECX.[11] = 1
		29:1	Reserved.	
		30	**Lock (R/W)**: If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
		31	**Debug Occurred (R/O)**: This “sticky bit” is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
		63:32	Reserved.	
C81H	3201	IA32_L3_QOS_CFG	**L3 QOS Configuration (R/W)**	If (CPUID.(EAX=10H, ECX=1):ECX.[2] = 1)
		0	**Enable (R/W)** Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode	
		63:1	Reserved.	
C8DH	3213	IA32_QM_EVTSEL	**Monitoring Event Select Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
		7:0	**Event ID:** ID of a supported monitoring event to report via IA32_QM_CTR.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address Hex	Register Address Decimal	Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
		31: 8	**Reserved**.	
		N+31:32	**Resource Monitoring ID:** ID for monitoring hardware to report monitored data via IA32_QM_CTR.	N = Ceil (Log_2 (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		63:N+32	**Reserved.**	
C8EH	3214	IA32_QM_CTR	**Monitoring Counter Register (R/O)**	If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
		61:0	**Resource Monitored Data**	
		62	**Unavailable**: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
		63	**Error:** If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
C8FH	3215	IA32_PQR_ASSOC	**Resource Association Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[12] =1) or (CPUID.(EAX=07H, ECX=0):EBX[15] =1))
		N-1:0	**Resource Monitoring ID (R/W):** ID for monitoring hardware to track internal operation, e.g. memory access.	N = Ceil (Log_2 (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		31:N	**Reserved**	
		63:32	**COS (R/W).** The class of service (COS) to enforce (on writes); returns the current COS when read.	If (CPUID.(EAX=07H, ECX=0):EBX.[15] = 1)
C90H - D8FH		Reserved MSR Address Space for CAT Mask Registers	See Section 17.18.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	
C90H	3216	IA32_L3_MASK_0	**L3 CAT Mask for COS0 (R/W)**	If (CPUID.(EAX=10H, ECX=0H):EBX[1] != 0)
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
C90H+ n	3216+n	IA32_L3_MASK_n	**L3 CAT Mask for COSn (R/W)**	n = CPUID.(EAX=10H, ECX=1H):EDX[15:0]
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D10H - D4FH		Reserved MSR Address Space for L2 CAT Mask Registers	See Section 17.18.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
D10H	3344	IA32_L2_MASK_0	**L2 CAT Mask for COS0 (R/W)**	If (CPUID.(EAX=10H, ECX=0H):EBX[2] != 0)
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D10H+n	3344+n	IA32_L2_MASK_n	**L2 CAT Mask for COSn (R/W)**	n = CPUID.(EAX=10H, ECX=2H):EDX[15:0]
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D90H	3472	IA32_BNDCFGS	**Supervisor State of MPX Configuration. (R/W)**	If (CPUID.(EAX=07H, ECX=0H):EBX[14] = 1)
		0	**EN:** Enable Intel MPX in supervisor mode	
		1	**BNDPRESERVE:** Preserve the bounds registers for near branch instructions in the absence of the BND prefix	
		11:2	Reserved, must be 0	
		63:12	**Base Address of Bound Directory.**	
DA0H	3488	IA32_XSS	**Extended Supervisor State Mask (R/W)**	If(CPUID.(0DH, 1):EAX.[3] = 1
		7:0	**Reserved**	
		8	**Trace Packet Configuration State (R/W)**	
		63:9	Reserved.	
DB0H	3504	IA32_PKG_HDC_CTL	**Package Level Enable/disable HDC (R/W)**	If CPUID.06H:EAX.[13] = 1
		0	**HDC_Pkg_Enable (R/W)** Force HDC idling or wake up HDC-idled logical processors in the package. See Section 14.5.2, "Package level Enabling HDC"	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB1H	3505	IA32_PM_CTL1	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[13] = 1
		0	**HDC_Allow_Block (R/W)** Allow/Block this logical processor for package level HDC control. See Section 14.5.3	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB2H	3506	IA32_THREAD_STALL	**Per-Logical_Processor HDC Idle Residency (R/0)**	If CPUID.06H:EAX.[13] = 1
		63:0	**Stall_Cycle_Cnt (R/W)** Stalled cycles due to HDC forced idle on this logical processor. See Section 14.5.4.1	If CPUID.06H:EAX.[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4000_ 0000H - 4000_ 00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSR in this range.	
C000_ 0080H		IA32_EFER	**Extended Feature Enables**	If (CPUID.80000001H:EDX.[20] || CPUID.80000001H:EDX.[29])
		0	**SYSCALL Enable: IA32_EFER.SCE (R/W)** Enables SYSCALL/SYSRET instructions in 64-bit mode.	
		7:1	Reserved.	
		8	**IA-32e Mode Enable: IA32_EFER.LME (R/W)** Enables IA-32e mode operation.	
		9	Reserved.	
		10	**IA-32e Mode Active: IA32_EFER.LMA (R)** Indicates IA-32e mode is active when set.	
		11	**Execute Disable Bit Enable: IA32_EFER.NXE (R/W)**	
		63:12	Reserved.	
C000_ 0081H		IA32_STAR	**System Call Target Address (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0082H		IA32_LSTAR	**IA-32e Mode System Call Target Address (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0084H		IA32_FMASK	**System Call Flag Mask (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0100H		IA32_FS_BASE	**Map of BASE Address of FS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0101H		IA32_GS_BASE	**Map of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0102H		IA32_KERNEL_GS_BASE	**Swap Target of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_ 0103H		IA32_TSC_AUX	Auxiliary TSC (RW)	If CPUID.80000001H: EDX[27] = 1
		31:0	AUX: Auxiliary signature of TSC	
		63:32	Reserved.	

NOTES:

1. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
2. The *_ADDR MSRs may or may not be present; this depends on flag settings in IA32_MC*i*_STATUS. See Section 15.3.2.3 and Section 15.3.2.4 for more information.
3. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have the CPUID signature DisplayFamily_DisplayModel of 06_17H.

The column "Shared/Unique" applies to multi-core processors based on Intel Core microarchitecture. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2.
17H	23	MSR_PLATFORM_ID	Shared	**Model Specific Platform ID (R)**
		7:0		Reserved.
		12:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		See Table 2-2.
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location." and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		5		Reserved.
		6		Reserved.
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		11		Intel TXT Capable Chipset. (R/O) 1 = Present; 0 = Not Present
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		13		Reserved.
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved.
		17:16		**APIC Cluster ID (R/O)**
		18		**N/2 Non-Integer Bus Ratio (R/O)** 0 = Integer ratio; 1 = Non-integer ratio
		19		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)**
		26:22		**Integer Bus Frequency Ratio (R/O)**

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
3AH	58	MSR_FEATURE_CONTROL	Unique	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		3	Unique	**SMRR Enable (R/WL)** When this bit is set and the lock bit is set makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Unique	**Last Branch Record 0 From IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Unique	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Unique	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Unique	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
A0H	160	MSR_SMRR_PHYSBASE	Unique	**System Management Mode Base Address register (WO in SMM)** Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		11:0		Reserved.
		31:12		PhysBase. SMRR physical Base Address.
		63:32		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
A1H	161	MSR_SMRR_PHYSMASK	Unique	**System Management Mode Physical Address Mask register (WO in SMM)** Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		10:0		Reserved.
		11		Valid. Physical address base and range mask are valid.
		31:12		PhysMask. SMRR physical address range mask.
		63:32		Reserved.
C1H	193	IA32_PMC0	Unique	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance Counter Register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Intel Core microarchitecture:
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333)
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.
		63:3		Reserved.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Enhanced Intel Core microarchitecture:
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333) ▪ 110B: 400 MHz (FSB 1600)

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
		11	Unique	**SMRR Capability Using MSR 0A0H and 0A1H (R)**
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	
		15:0		Current Performance State Value.
		30:16		Reserved.
		31		XE Operation (R/O). If set, XE operation is enabled. Default is cleared.
		39:32		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		45		Reserved.
		46		Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.
		63:47		Reserved.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
19DH	413	MSR_THERM2_CTL	Unique	
		15:0		Reserved.
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.
		8		Reserved.
		9		**Hardware Prefetcher Disable (R/W)** When set, disables the hardware prefetcher operation on streams of data. When clear (default), enables the prefetch queue. Disabling of the hardware prefetcher may impact processor performance.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Shared	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19	Shared	**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Shared	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		36:35		Reserved.
		37	Unique	**DCU Prefetcher Disable (R/W)** When set to 1, The DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.
		38	Shared	**IDA Disable (R/W)** When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of IDA. If power-on default value is 1, IDA is available in the processor. If power-on default value is 0, IDA is not available.
		39	Unique	**IP Prefetcher Disable (R/W)** When set to 1, The IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.
		63:40		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
200H	512	IA32_MTRR_PHYSBASE0	Unique	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Unique	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Unique	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Unique	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Unique	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Unique	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Unique	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Unique	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Unique	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Unique	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Unique	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Unique	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Unique	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Unique	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Unique	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Unique	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Unique	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Unique	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Unique	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Unique	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Unique	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Unique	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Unique	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Unique	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Unique	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Unique	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Unique	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
309H	777	MSR_PERF_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)**

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30AH	778	IA32_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30AH	778	MSR_PERF_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)**
30BH	779	IA32_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
30BH	779	MSR_PERF_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)**
345H	837	IA32_PERF_CAPABILITIES	Unique	See Table 2-2. See Section 17.4.1, “IA32_DEBUGCTL MSR.”
345H	837	MSR_PERF_CAPABILITIES	Unique	RO. This applies to processors that do not support architectural perfmon version 2.
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		63:8		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38DH	909	MSR_PERF_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)**
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
38EH	910	MSR_PERF_GLOBAL_STATUS	Unique	See Section 18.4.2, “Global Counter Control Facilities.”
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
38FH	911	MSR_PERF_GLOBAL_CTRL	Unique	See Section 18.4.2, “Global Counter Control Facilities.”
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Unique	See Section 18.4.2, “Global Counter Control Facilities.”
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.4.4, “Processor Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MC0_CTL	Unique	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC3_MISC	Unique	
414H	1044	IA32_MC5_CTL	Unique	
415H	1045	IA32_MC5_STATUS	Unique	
416H	1046	IA32_MC5_ADDR	Unique	
417H	1047	IA32_MC5_MISC	Unique	
419H	1045	IA32_MC6_STATUS	Unique	Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Chapter 23.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, “Debug Store (DS) Mechanism.”
107CC H		MSR_EMON_L3_CTR_CTL0	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CD H		MSR_EMON_L3_CTR_CTL1	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CE H		MSR_EMON_L3_CTR_CTL2	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CF H		MSR_EMON_L3_CTR_CTL3	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H		MSR_EMON_L3_CTR_CTL4	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H		MSR_EMON_L3_CTR_CTL5	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H		MSR_EMON_L3_CTR_CTL6	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H		MSR_EMON_L3_CTR_CTL7	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D8 H		MSR_EMON_L3_GL_CTL	Unique	**L3/FSB Common Control Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
C000_ 0080H		IA32_EFER	Unique	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
C000_ 0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

2.3 MSRS IN THE 45 NM AND 32 NM INTEL® ATOM™ PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1CH, 06_26H, 06_27H, 06_35H and 06_36H; see Table 2-1.

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

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_ SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_ COUNTER	Unique	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2.
17H	23	MSR_PLATFORM_ID	Shared	**Model Specific Platform ID (R)**
		7:0		Reserved.
		12:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		63:13		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0		Reserved.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		3		**AERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		4		**BERR# Enable for initiator bus requests (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		5		Reserved.
		6		Reserved.
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		8		Reserved.
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**AERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0.
		11		Reserved.
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0.
		13		**Reserved.**
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		**APIC Cluster ID (R/O)** Always 00B.
		19: 18		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)** Always 00B.
		26:22		**Integer Bus Frequency Ratio (R/O)**
3AH	58	IA32_FEATURE_CONTROL	Unique	**Control Features in Intel 64Processor (R/W)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Unique	**Last Branch Record 0 From IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction** . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_ LASTBRANCH_4_FROM_IP	Unique	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_ LASTBRANCH_5_FROM_IP	Unique	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_ LASTBRANCH_6_FROM_IP	Unique	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_ LASTBRANCH_7_FROM_IP	Unique	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Unique	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Unique	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Unique	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_ LASTBRANCH_4_TO_IP	Unique	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_ LASTBRANCH_5_TO_IP	Unique	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_ LASTBRANCH_6_TO_IP	Unique	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_ LASTBRANCH_7_TO_IP	Unique	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Shared	**BIOS Update Trigger Register (W)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Unique	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance Counter Register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Intel Atom microarchitecture:
		2:0		▪ 111B: 083 MHz (FSB 333) ▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Shared	**Memory Type Range Register (R)** See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled. (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	
		15:0		Current Performance State Value.
		39:16		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		63:45		Reserved.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2.
19DH	413	MSR_THERM2_CTL	Shared	
		15:0		Reserved.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:17		Reserved.
1A0H	416	IA32_MISC_ENABLE	Unique	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 0.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.
		8		Reserved.
		9		Reserved.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Shared	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19		Reserved.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Unique	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	IA32_MTRR_PHYSBASE0	Shared	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Shared	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Shared	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Shared	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Shared	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Shared	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Shared	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Shared	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Shared	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Shared	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Shared	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Shared	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Shared	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Shared	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Shared	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Shared	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Shared	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Shared	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Shared	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Shared	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Shared	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Shared	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Shared	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Shared	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Shared	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Shared	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Shared	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30BH	779	IA32_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Shared	See Table 2-2. See Section 17.4.1, “IA32_DEBUGCTL MSR.”
38DH	909	IA32_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.4.4, “Processor Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MC0_CTL	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
402H	1026	IA32_MC0_ADDR	Shared	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
405H	1029	IA32_MC1_STATUS	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
408H	1032	IA32_MC2_CTL	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
409H	1033	IA32_MC2_STATUS	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40AH	1034	IA32_MC2_ADDR	Shared	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	IA32_MC3_STATUS	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40EH	1038	IA32_MC3_ADDR	Shared	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
411H	1041	IA32_MC4_STATUS	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
412H	1042	IA32_MC4_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, "Debug Store (DS) Mechanism."
C000_ 0080H		IA32_EFER	Unique	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor with the CPUID signature with DisplayFamily_DisplayModel of 06_27H.

Table 2-5. MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C2_RESIDENCY	Package	**Package C2 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C2 Residency Counter. (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.
3F9H	1017	MSR_PKG_C4_RESIDENCY	Package	**Package C4 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C4 Residency Counter. (R/O) Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.

Table 2-5. MSRs Supported by Intel® Atom™ Processors (Contd.)with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	**Package C6 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C6 Residency Counter. (R/O) Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.

2.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 2-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_37H, 06_4AH, 06_4DH, 06_5AH, and 06_5DH; see Table 2-1. The MSRs listed in Table 2-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 2-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 2-8, Table 2-9, and Table 2-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
1BH	27	IA32_APIC_BASE	Core	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Module	**Processor Hard Power-On Configuration (R/W)** Writes ignored.
		63:0		Reserved **(R/O)**
34H	52	MSR_SMI_COUNT	Core	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Running count of SMI events since last RESET.
		63:32		Reserved.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Core	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Core	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Core	**Performance Counter Register** See Table 2-2.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include 111b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Core	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Core	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	**Memory Type Range Register (R)** See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
174H	372	IA32_SYSENTER_CS	Core	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Core	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Core	See Table 2-2.
179H	377	IA32_MCG_CAP	Core	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Core	
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Core	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**Reserved**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Core	See Table 2-2.
198H	408	IA32_PERF_STATUS	Module	See Table 2-2.
199H	409	IA32_PERF_CTL	Core	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Core	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		**Temperature Target (R)** The default thermal throttling or PROCHOT# activation temperature in degree C, The effective temperature for thermal throttling or PROCHOT# activation is “Temperature Target” + “Target Offset”
		29:24		**Target Offset (R/W)** Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).
		63:30		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Module	**Offcore Response Event Select Register (R/W)**
1A7H	423	MSR_OFFCORE_RSP_1	Module	**Offcore Response Event Select Register (R/W)**
1B0H	432	IA32_ENERGY_PERF_BIAS	Core	See Table 2-2.
1D9H	473	IA32_DEBUGCTL	Core	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Core	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Core	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Core	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Core	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Core	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Core	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Core	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Core	See Table 2-2. See Section 18.4.2, "Global Counter Control Facilities."
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.
400H	1024	IA32_MC0_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Core	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Core	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Core	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Core	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Core	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Core	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Core	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address Hex	Address Dec	Register Name	Scope	Bit Description
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Core	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
491H	1169	IA32_VMX_FMFUNC	Core	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Core	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Core	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, "Debug Store (DS) Mechanism."
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C1 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C1 states. Counts at the TSC frequency.
6E0H	1760	IA32_TSC_DEADLINE	Core	**TSC Target of Local APIC's TSC Deadline Mode (R/W)** See Table 2-2
C000_0080H		IA32_EFER	Core	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Core	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Core	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Core	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Core	**Map of BASE Address of FS (R/W)** See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_ 0101H		IA32_GS_BASE	Core	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Core	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0103H		IA32_TSC_AUX	Core	**AUXILIARY TSC Signature. (R/W)** See Table 2-2

Table 2-7 lists model-specific registers (MSRs) that are common to Intel® Atom™ processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		7:0		Reserved.
		13:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		**See** Table 2-2
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Reserved**
		2		**Enable VMX outside SMX operation (R/WL)**
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5 and record format in Section 17.4.8.1
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Core	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Core	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_ LASTBRANCH_4_FROM_IP	Core	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
45H	69	MSR_ LASTBRANCH_5_FROM_IP	Core	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_ LASTBRANCH_6_FROM_IP	Core	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_ LASTBRANCH_7_FROM_IP	Core	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Core	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Core	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Core	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_ LASTBRANCH_4_TO_IP	Core	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_ LASTBRANCH_5_TO_IP	Core	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_ LASTBRANCH_6_TO_IP	Core	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_ LASTBRANCH_7_TO_IP	Core	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
CDH	205	MSR_FSB_FREQ	Module	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture:
		2:0		▪ 100B: 080.0 MHz ▪ 000B: 083.3 MHz ▪ 001B: 100.0 MHz ▪ 010B: 133.3 MHz ▪ 011B: 116.7 MHz
		63:3		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only).
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
11EH	281	MSR_BBL_CR_CTL3	Module	
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled. (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Core	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Module	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 0.
		6:4		Reserved.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Module	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Module	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Module	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.8.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Core	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
38EH	910	IA32_PERF_GLOBAL_STATUS	Core	See Table 2-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Core	See Table 2-2. See Section 18.4.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.4.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS for precise event on IA32_PMC0. (R/W)
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	**Module C6 Residency Counter (R/0)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_37H) and Intel Atom processors (CPUID signatures with DisplayFamily_DisplayModel of 06_4AH, 06_5AH, 06_5DH).

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)**
		14:0		Package Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		23:17		Time Window for Power Limit #1. (R/W) in unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.
		63:24		Reserved
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8

Table 2-9 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_37H).

Table 2-9. Specific MSRs Supported by Intel® Atom™ Processor E3000 Series with CPUID Signature 06_37H

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
668H	1640	MSR_CC6_DEMOTION_POLICY_CONFIG	Package	**Core C6 demotion policy config MSR**
		63:0		Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.
669H	1641	MSR_MC6_DEMOTION_POLICY_CONFIG	Package	**Module C6 demotion policy config MSR**
		63:0		Controls module (i.e. two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	**Module C6 Residency Counter (R/0)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

Table 2-10 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor C2000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_4DH).

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series with CPUID Signature 06_4DH

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
1A4H	420	MSR_MISC_FEATURE_CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		**Reserved**
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode (RW)**
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
66EH	1646	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameter (R/O)**
		14:0		Thermal Spec Power. (R/0) The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT
		63:15		Reserved

2.4.2 MSRs In Intel Atom Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID signature with DisplayFamily_DisplayModel including 06_4CH; see Table 2-1.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture

Address Hex	Address Dec	Register Name	Scope	Bit Description
CDH	205	MSR_FSB_FREQ	Module	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Airmont microarchitecture:
		3:0		▪ 0000B: 083.3 MHz ▪ 0001B: 100.0 MHz ▪ 0010B: 133.3 MHz ▪ 0011B: 116.7 MHz ▪ 0100B: 080.0 MHz ▪ 0101B: 093.3 MHz ▪ 0110B: 090.0 MHz ▪ 0111B: 088.9 MHz ▪ 1000B: 087.5 MHz
		63:5		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - Deep Power Down Technology is the max C-State 010b - C7 is the max C-State to include
		63:19		Reserved.
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)**
		14:0		PP0 Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains."
		16		Reserved
		23:17		Time Window for Power Limit #1. (R/W) Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.
		63:24		Reserved

2.5 MSRS IN NEXT GENERATION INTEL ATOM PROCESSORS

Next Generation Intel Atom processors are based on the Goldmont microarchitecture. These processors support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID signature with DisplayFamily_DisplayModel including 06_5CH; see Table 2-1.

In the Goldmont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture

Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		49:0		Reserved.
		52:50		See Table 2-2.
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		18		**SGX global functions enable (R/WL)**
		63:19		Reserved.
3BH	59	IA32_TSC_ADJUST	Core	**Per-Core TSC ADJUST (R/W)** See Table 2-2.
C3H	195	IA32_PMC2	Core	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Core	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
17DH	381	MSR_SMM_MCA_CAP	Core	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
188H	392	IA32_PERFEVTSEL2	Core	See Table 2-2.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
189H	393	IA32_PERFEVTSEL3	Core	See Table 2-2.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Core	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Package	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 1.
		6:4		Reserved.
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Package	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		**Reserved**
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved.
1AAH	426	MSR_MISC_PWR_MGMT	Package	See http://biosbits.org.
		0		**EIST Hardware Coordination Disable (R/W)** When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		21:1		Reserved.
		22		**Thermal Interrupt Coordination Enable (R/W)** If set, then thermal interrupt on one core is routed to all cores.
		63:23		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode by Core Groups (RW)** **Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically.** **For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.**
		7:0	Package	**Maximum Ratio Limit for Active cores in Group 0** Maximum turbo ratio limit when number of active cores is less or equal to Group 0 threshold.
		15:8	Package	**Maximum Ratio Limit for Active cores in Group 1** Maximum turbo ratio limit when number of active cores is less or equal to Group 1 threshold and greater than Group 0 threshold.
		23:16	Package	**Maximum Ratio Limit for Active cores in Group 2** Maximum turbo ratio limit when number of active cores is less or equal to Group 2 threshold and greater than Group 1 threshold.
		31:24	Package	**Maximum Ratio Limit for Active cores in Group 3** Maximum turbo ratio limit when number of active cores is less or equal to Group 3 threshold and greater than Group 2 threshold.
		39:32	Package	**Maximum Ratio Limit for Active cores in Group 4** Maximum turbo ratio limit when number of active cores is less or equal to Group 4 threshold and greater than Group 3 threshold.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Ratio Limit for Active cores in Group 5** Maximum turbo ratio limit when number of active cores is less or equal to Group 5 threshold and greater than Group 4 threshold.
		55:48	Package	**Maximum Ratio Limit for Active cores in Group 6** Maximum turbo ratio limit when number of active cores is less or equal to Group 6 threshold and greater than Group 5 threshold.
		63:56	Package	**Maximum Ratio Limit for Active cores in Group 7** Maximum turbo ratio limit when number of active cores is less or equal to Group 7 threshold and greater than Group 6 threshold.
1AEH	430	MSR_TURBO_GROUP_CORE CNT	Package	**Group Size of Active Cores for Turbo Mode Operation (RW)** **Writes of 0 threshold is ignored**
		7:0	Package	**Group 0 Core Count Threshold** Maximum number of active cores to operate under Group 0 Max Turbo Ratio limit.
		15:8	Package	**Group 1 Core Count Threshold** Maximum number of active cores to operate under Group 1 Max Turbo Ratio limit. Must be greater than Group 0 Core Count.
		23:16	Package	**Group 2 Core Count Threshold** Maximum number of active cores to operate under Group 2 Max Turbo Ratio limit. Must be greater than Group 1 Core Count.
		31:24	Package	**Group 3 Core Count Threshold** Maximum number of active cores to operate under Group 3 Max Turbo Ratio limit. Must be greater than Group 2 Core Count.
		39:32	Package	**Group 4 Core Count Threshold** Maximum number of active cores to operate under Group 4 Max Turbo Ratio limit. Must be greater than Group 3 Core Count.
		47:40	Package	**Group 5 Core Count Threshold** Maximum number of active cores to operate under Group 5 Max Turbo Ratio limit. Must be greater than Group 4 Core Count.
		55:48	Package	**Group 6 Core Count Threshold** Maximum number of active cores to operate under Group 6 Max Turbo Ratio limit. Must be greater than Group 5 Core Count.
		63:56	Package	**Group 7 Core Count Threshold** Maximum number of active cores to operate under Group 7 Max Turbo Ratio limit. Must be greater than Group 6 Core Count and not less than the total number of processor cores in the package. E.g. specify 255.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.8.2, “Filtering of Last Branch Records.”
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		9		**EN_CALL_STACK**
		63:10		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Core	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-4) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
210H	528	IA32_MTRR_PHYSBASE8	Core	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Core	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Core	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Core	See Table 2-2.
280H	640	IA32_MC0_CTL2	Module	See Table 2-2.
281H	641	IA32_MC1_CTL2	Module	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Module	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
300H	768	MSR_SGXOWNEREPOCH0	Package	**Lower 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	769	MSR_SGXOWNEREPOCH1	Package	**Upper 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.
38EH	910	IA32_PERF_GLOBAL_ STATUS	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		**Ovf_PMC0**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**
		33		**Ovf_FixedCtr1**
		34		**Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Trace_ToPA_PMI.**
		57:56		Reserved.
		58		**LBR_Frz.**
		59		**CTR_Frz.**
		60		**ASCI.**
		61		**Ovf_Uncore**
		62		**Ovf_BufDSSAVE**
		63		**CondChgd**
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Core	See Table 2-2. See Section 18.2.4, “Architectural Performance Monitoring Version 4.”
		0		**Set 1 to clear Ovf_PMC0**
		1		**Set 1 to clear Ovf_PMC1**
		2		**Set 1 to clear Ovf_PMC2**
		3		**Set 1 to clear Ovf_PMC3**
		31:4		Reserved.
		32		**Set 1 to clear Ovf_FixedCtr0**
		33		**Set 1 to clear Ovf_FixedCtr1**
		34		**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Set 1 to clear Trace_ToPA_PMI.**
		57:56		Reserved.
		58		**Set 1 to clear LBR_Frz.**
		59		**Set 1 to clear CTR_Frz.**
		60		**Set 1 to clear ASCI.**
		61		**Set 1 to clear Ovf_Uncore**
		62		**Set 1 to clear Ovf_BufDSSAVE**
		63		**Set 1 to clear CondChgd**
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Core	See Table 2-2. See Section 18.2.4, “Architectural Performance Monitoring Version 4.”

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		**Set 1 to cause Ovf_PMC0 = 1**
		1		**Set 1 to cause Ovf_PMC1 = 1**
		2		**Set 1 to cause Ovf_PMC2 = 1**
		3		**Set 1 to cause Ovf_PMC3 = 1**
		31:4		Reserved.
		32		**Set 1 to cause Ovf_FixedCtr0 = 1**
		33		**Set 1 to cause Ovf_FixedCtr1 = 1**
		34		**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55		**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.
		58		**Set 1 to cause LBR_Frz = 1**
		59		**Set 1 to cause CTR_Frz = 1**
		60		**Set 1 to cause ASCI = 1**
		61		**Set 1 to cause Ovf_Uncore**
		62		**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	914	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.4.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
406H	1030	IA32_MC1_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C3H	1219	IA32_A_PMC2	Core	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Core	See Table 2-2.
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Core	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, “Interactions with Authenticated Code Modules (ACMs)”
		15:1		Reserved.
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, “Interactions with Authenticated Code Modules (ACMs)”
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Core	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Core	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Core	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
571H	1393	IA32_RTIT_STATUS	Core	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ
		48:32		**PacketByteCnt**
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Core	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Core	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Core	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Core	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Core	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in Watts) is in unit of, 1W/2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in Joules) is in unit of, 1Joule/ (2^ESU); where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.
		15:13		Reserved
		19:16		Time Unit. Time related information (in seconds) is in unit of, 1S/2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.
		63:20		Reserved

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-18 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC_IRTL1	Package	**Package C6/C7S Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-18 for supported time unit encodings
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-18 for supported time unit encodings

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		**Package C2 Residency Counter. (R/O)** Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)**
		14:0		**Thermal Spec Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		15		Reserved.
		30:16		**Minimum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		31		Reserved.
		46:32		**Maximum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		47		Reserved.
		54:48		**Maximum Time Window (R/W)** Specified by 2^Y * (1.0 + Z/4.0) * Time_Unit, where "Y" is the unsigned integer value represented. by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT
		63:55		Reserved.
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names,
		63:0		**Package C10 Residency Counter. (R/O)** Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		3		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		8:4		Reserved.
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		11		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		12		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		14		**Maximum Efficiency Frequency Status (R0)** When set, frequency is reduced below the maximum efficiency frequency.
		15		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24:20		Reserved.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		28		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Maximum Efficiency Frequency Log** When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:31		Reserved.
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction** . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.6 and record format in Section 17.4.8.1
		0:47		**From Linear Address (R/W)**
		62:48		Signed extension of bits 47:0.
		63		Mispred
681H	1665	MSR_ LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_ LASTBRANCH_2_FROM_IP	Core	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_ LASTBRANCH_3_FROM_IP	Core	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_ LASTBRANCH_4_FROM_IP	Core	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_ LASTBRANCH_5_FROM_IP	Core	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_ LASTBRANCH_6_FROM_IP	Core	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Core	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Core	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Core	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Core	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Core	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Core	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Core	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Core	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Core	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
690H	1680	MSR_ LASTBRANCH_16_FROM_IP	Core	**Last Branch Record 16 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
691H	1681	MSR_ LASTBRANCH_17_FROM_IP	Core	**Last Branch Record 17 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_ LASTBRANCH_18_FROM_IP	Core	**Last Branch Record 18 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_ LASTBRANCH_19_FROM_IP	Core	**Last Branch Record 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Core	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Core	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Core	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Core	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Core	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Core	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Core	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Core	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Core	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Core	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Core	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Core	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the **Destination instruction** and elapsed cycles from last LBR update. See also: ▪ Section 17.6
		0:47		**Target Linear Address (R/W)**
		63:48		Elapsed cycles from last update to the LBR.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Core	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Core	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Core	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Core	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Core	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Core	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Core	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Core	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Core	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Core	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Core	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Core	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Core	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Core	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Core	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
6D0H	1744	MSR_ LASTBRANCH_16_TO_IP	Core	**Last Branch Record 16 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D1H	1745	MSR_ LASTBRANCH_17_TO_IP	Core	**Last Branch Record 17 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_ LASTBRANCH_18_TO_IP	Core	**Last Branch Record 18 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_ LASTBRANCH_19_TO_IP	Core	**Last Branch Record 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Core	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Core	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Core	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Core	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Core	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Core	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Core	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Core	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Core	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Core	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Core	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Core	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Core	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Core	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Core	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Core	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Core	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Core	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Core	x2APIC Spurious Interrupt Vector register (R/W)

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
810H	2064	IA32_X2APIC_ISR0	Core	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Core	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Core	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Core	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Core	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Core	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Core	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Core	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Core	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Core	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Core	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Core	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Core	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Core	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Core	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Core	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Core	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Core	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Core	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Core	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Core	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Core	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Core	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Core	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Core	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Core	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Core	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Core	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Core	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Core	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Core	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Core	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Core	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Core	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Core	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Core	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Core	x2APIC Self IPI register (W/O)

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
C8FH	3215	IA32_PQR_ASSOC	Core	**Resource Association Register (R/W)**
		31:0		**Reserved**
		33:32		**COS (R/W).**
		63: 34		**Reserved**
D10H	3344	IA32_L2_QOS_MASK_0	Module	**L2 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D11H	3345	IA32_L2_QOS_MASK_1	Module	**L2 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D12H	3346	IA32_L2_QOS_MASK_2	Module	**L2 Class Of Service Mask - COS 2 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D13H	3347	IA32_L2_QOS_MASK_3	Package	**L2 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L2 ways for COS 3 enforcement**
		63:20		**Reserved**
D90H	3472	IA32_BNDCFGS	Core	See Table 2-2.
DA0H	3488	IA32_XSS	Core	See Table 2-2.
See Table 2-6, and Table 2-12 for MSR definitions applicable to processors with CPUID signature 06_5CH.				

2.6 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 2-13 lists model-specific registers (MSRs) that are common for Intel® microarchitecture code name Nehalem. These include Intel Core i7 and i5 processor family. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1AH, 06_1EH, 06_1FH, 06_2EH, see Table 2-1. Additional MSRs specific to 06_1AH, 06_1EH, 06_1FH are listed in Table 2-14. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at http://biosbits.org.

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

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_ SIZE	Thread	See Section 8.10.5, “Monitor/Mwait Address Range Determination,” and Table 2-2.
10H	16	IA32_TIME_ STAMP_COUNTER	Thread	See Section 17.16, “Time-Stamp Counter,” and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
17H	23	MSR_PLATFORM_ID	Package	**Model Specific Platform ID (R)**
		49:0		Reserved.
		52:50		See Table 2-2.
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, “Local APIC Status and Location,” and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Running count of SMI events since last RESET.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
79H	121	IA32_BIOS_ UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_ SIGN_ID	Thread	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Thread	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Thread	**Performance Counter Register** See Table 2-2.
C3H	195	IA32_PMC2	Thread	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Thread	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	see http://biosbits.org.
		7:0		Reserved.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDC-TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDC/TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 133.33MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16		Reserved.
		24		**Interrupt filtering enable (R/W)** When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Thread	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Core	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		0		Reserved.
		3:1		**On demand Clock Modulation Duty Cycle (R/W)**
		4		**On demand Clock Modulation Enable (R/W)**
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Thread	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 1.
		6:4		Reserved.
		7	Thread	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Thread	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Thread	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_TEMPERATURE_TARGET	Thread	
		15:0		Reserved.
		23:16		**Temperature Target (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		See http://biosbits.org.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0	Package	**EIST Hardware Coordination Disable (R/W)** When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		1	Thread	**Energy/Performance Bias Enable (R/W)** This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].
		63:2		Reserved.
1ACH	428	MSR_TURBO_POWER_ CURRENT_LIMIT		See http://biosbits.org.
		14:0	Package	**TDP Limit (R/W)** TDP limit in 1/8 Watt granularity.
		15	Package	**TDP Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		30:16	Package	**TDC Limit (R/W)** TDC limit in 1/8 Amp granularity.
		31	Package	**TDC Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.8.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to ‘1’, will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Package	See Table 2-2.
281H	641	IA32_MC1_CTL2	Package	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Core	See Table 2-2.
285H	645	IA32_MC5_CTL2	Core	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, “IA32_DEBUGCTL MSR.”
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		11:8		PEBS_REC_FORMAT. See Table 2-2.
		12		SMM_FREEZE. See Table 2-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
38EH	910	MSR_PERF_GLOBAL_STATUS	Thread	**(RO)**
		61		**UNC_Ovf** Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Thread	**(R/W)**
		61		**CLR_UNC_Ovf** Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.8.1.1, “Processor Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.8.1.2, “Load Latency Performance Monitoring Facility.”
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
402H	1026	IA32_MC0_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
413H	1043	IA32_MC4_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
414H	1044	IA32_MC5_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
415H	1045	IA32_MC5_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
416H	1046	IA32_MC5_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
417H	1047	IA32_MC5_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, “Basic VMX Information.”
481H	1153	IA32_VMX_PINBASED_ CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, “VM-Execution Controls.”
482H	1154	IA32_VMX_PROCBASED_ CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, “VM-Execution Controls.”
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, “VM-Exit Controls.”
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, “VM-Entry Controls.”
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, “Miscellaneous Data.”

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O).** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: • Last Branch Record Stack TOS at 1C9H • Section 17.8.1 and record format in Section 17.4.8.1
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Thread	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Thread	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Thread	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Thread	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

Table 2-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_ 0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W)** See Table 2-2 and Section 17.16.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.6.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

Intel Xeon Processor 5500 and 3400 series support additional model-specific registers listed in Table 2-14. These MSRs also apply to Intel Core i7 and i5 processor family CPUID signature with DisplayFamily_DisplayModel of 06_1AH, 06_1EH and 06_1FH, see Table 2-1.

Table 2-14. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (not available to model 06_2EH)
		7:0		**Maximum Turbo Ratio Limit 1C (R/O)** Maximum Turbo mode ratio limit with 1 core active.
		15:8		**Maximum Turbo Ratio Limit 2C (R/O)** Maximum Turbo mode ratio limit with 2cores active.
		23:16		**Maximum Turbo Ratio Limit 3C (R/O)** Maximum Turbo mode ratio limit with 3cores active.
		31:24		**Maximum Turbo Ratio Limit 4C (R/O)** Maximum Turbo mode ratio limit with 4 cores active.
		63:32		Reserved.
301H	769	MSR_GQ_SNOOP_MESF	Package	
		0		**From M to S (R/W)**
		1		**From E to S (R/W)**
		2		**From S to S (R/W)**
		3		**From F to S (R/W)**
		4		**From M to I (R/W)**
		5		**From E to I (R/W)**
		6		**From S to I (R/W)**
		7		**From F to I (R/W)**
		63:8		Reserved.
391H	913	MSR_UNCORE_PERF_ GLOBAL_CTRL	Package	See Section 18.8.2.1, "Uncore Performance Monitoring Management Facility."
392H	914	MSR_UNCORE_PERF_ GLOBAL_STATUS	Package	See Section 18.8.2.1, "Uncore Performance Monitoring Management Facility."
393H	915	MSR_UNCORE_PERF_ GLOBAL_OVF_CTRL	Package	See Section 18.8.2.1, "Uncore Performance Monitoring Management Facility."

Table 2-14. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNCORE_FIXED_CTR0	Package	See Section 18.8.2.1, "Uncore Performance Monitoring Management Facility."
395H	917	MSR_UNCORE_FIXED_CTR_CTRL	Package	See Section 18.8.2.1, "Uncore Performance Monitoring Management Facility."
396H	918	MSR_UNCORE_ADDR_OPCODE_MATCH	Package	See Section 18.8.2.3, "Uncore Address/Opcode Match MSR."
3B0H	960	MSR_UNCORE_PMC0	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B1H	961	MSR_UNCORE_PMC1	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B2H	962	MSR_UNCORE_PMC2	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B3H	963	MSR_UNCORE_PMC3	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B4H	964	MSR_UNCORE_PMC4	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B6H	966	MSR_UNCORE_PMC6	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3B7H	967	MSR_UNCORE_PMC7	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C0H	944	MSR_UNCORE_PERFEVTSEL0	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C1H	945	MSR_UNCORE_PERFEVTSEL1	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C2H	946	MSR_UNCORE_PERFEVTSEL2	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C3H	947	MSR_UNCORE_PERFEVTSEL3	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C4H	948	MSR_UNCORE_PERFEVTSEL4	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C5H	949	MSR_UNCORE_PERFEVTSEL5	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C6H	950	MSR_UNCORE_PERFEVTSEL6	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."
3C7H	951	MSR_UNCORE_PERFEVTSEL7	Package	See Section 18.8.2.2, "Uncore Performance Event Configuration Facility."

2.6.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 2-13 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-15. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2EH.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
394H	816	MSR_W_PMON_FIXED_CTR	Package	Uncore W-box perfmon fixed counter
395H	817	MSR_W_PMON_FIXED_ CTR_CTL	Package	Uncore U-box perfmon fixed counter control MSR
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
451H	1105	IA32_MC20_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
452H	1106	IA32_MC20_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
453H	1107	IA32_MC20_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
455H	1109	IA32_MC21_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
456H	1110	IA32_MC21_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
457H	1111	IA32_MC21_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
C00H	3072	MSR_U_PMON_GLOBAL_CTRL	Package	Uncore U-box perfmon global control MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C01H	3073	MSR_U_PMON_GLOBAL_STATUS	Package	Uncore U-box perfmon global status MSR.
C02H	3074	MSR_U_PMON_GLOBAL_OVF_CTRL	Package	Uncore U-box perfmon global overflow control MSR.
C10H	3088	MSR_U_PMON_EVNT_SEL	Package	Uncore U-box perfmon event select MSR.
C11H	3089	MSR_U_PMON_CTR	Package	Uncore U-box perfmon counter MSR.
C20H	3104	MSR_B0_PMON_BOX_CTRL	Package	Uncore B-box 0 perfmon local box control MSR.
C21H	3105	MSR_B0_PMON_BOX_STATUS	Package	Uncore B-box 0 perfmon local box status MSR.
C22H	3106	MSR_B0_PMON_BOX_OVF_CTRL	Package	Uncore B-box 0 perfmon local box overflow control MSR.
C30H	3120	MSR_B0_PMON_EVNT_SEL0	Package	Uncore B-box 0 perfmon event select MSR.
C31H	3121	MSR_B0_PMON_CTR0	Package	Uncore B-box 0 perfmon counter MSR.
C32H	3122	MSR_B0_PMON_EVNT_SEL1	Package	Uncore B-box 0 perfmon event select MSR.
C33H	3123	MSR_B0_PMON_CTR1	Package	Uncore B-box 0 perfmon counter MSR.
C34H	3124	MSR_B0_PMON_EVNT_SEL2	Package	Uncore B-box 0 perfmon event select MSR.
C35H	3125	MSR_B0_PMON_CTR2	Package	Uncore B-box 0 perfmon counter MSR.
C36H	3126	MSR_B0_PMON_EVNT_SEL3	Package	Uncore B-box 0 perfmon event select MSR.
C37H	3127	MSR_B0_PMON_CTR3	Package	Uncore B-box 0 perfmon counter MSR.
C40H	3136	MSR_S0_PMON_BOX_CTRL	Package	Uncore S-box 0 perfmon local box control MSR.
C41H	3137	MSR_S0_PMON_BOX_STATUS	Package	Uncore S-box 0 perfmon local box status MSR.
C42H	3138	MSR_S0_PMON_BOX_OVF_CTRL	Package	Uncore S-box 0 perfmon local box overflow control MSR.
C50H	3152	MSR_S0_PMON_EVNT_SEL0	Package	Uncore S-box 0 perfmon event select MSR.
C51H	3153	MSR_S0_PMON_CTR0	Package	Uncore S-box 0 perfmon counter MSR.
C52H	3154	MSR_S0_PMON_EVNT_SEL1	Package	Uncore S-box 0 perfmon event select MSR.
C53H	3155	MSR_S0_PMON_CTR1	Package	Uncore S-box 0 perfmon counter MSR.
C54H	3156	MSR_S0_PMON_EVNT_SEL2	Package	Uncore S-box 0 perfmon event select MSR.
C55H	3157	MSR_S0_PMON_CTR2	Package	Uncore S-box 0 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C56H	3158	MSR_S0_PMON_EVNT_SEL3	Package	Uncore S-box 0 perfmon event select MSR.
C57H	3159	MSR_S0_PMON_CTR3	Package	Uncore S-box 0 perfmon counter MSR.
C60H	3168	MSR_B1_PMON_BOX_CTRL	Package	Uncore B-box 1 perfmon local box control MSR.
C61H	3169	MSR_B1_PMON_BOX_STATUS	Package	Uncore B-box 1 perfmon local box status MSR.
C62H	3170	MSR_B1_PMON_BOX_OVF_CTRL	Package	Uncore B-box 1 perfmon local box overflow control MSR.
C70H	3184	MSR_B1_PMON_EVNT_SEL0	Package	Uncore B-box 1 perfmon event select MSR.
C71H	3185	MSR_B1_PMON_CTR0	Package	Uncore B-box 1 perfmon counter MSR.
C72H	3186	MSR_B1_PMON_EVNT_SEL1	Package	Uncore B-box 1 perfmon event select MSR.
C73H	3187	MSR_B1_PMON_CTR1	Package	Uncore B-box 1 perfmon counter MSR.
C74H	3188	MSR_B1_PMON_EVNT_SEL2	Package	Uncore B-box 1 perfmon event select MSR.
C75H	3189	MSR_B1_PMON_CTR2	Package	Uncore B-box 1 perfmon counter MSR.
C76H	3190	MSR_B1_PMON_EVNT_SEL3	Package	Uncore B-box 1vperfmon event select MSR.
C77H	3191	MSR_B1_PMON_CTR3	Package	Uncore B-box 1 perfmon counter MSR.
C80H	3120	MSR_W_PMON_BOX_CTRL	Package	Uncore W-box perfmon local box control MSR.
C81H	3121	MSR_W_PMON_BOX_STATUS	Package	Uncore W-box perfmon local box status MSR.
C82H	3122	MSR_W_PMON_BOX_OVF_CTRL	Package	Uncore W-box perfmon local box overflow control MSR.
C90H	3136	MSR_W_PMON_EVNT_SEL0	Package	Uncore W-box perfmon event select MSR.
C91H	3137	MSR_W_PMON_CTR0	Package	Uncore W-box perfmon counter MSR.
C92H	3138	MSR_W_PMON_EVNT_SEL1	Package	Uncore W-box perfmon event select MSR.
C93H	3139	MSR_W_PMON_CTR1	Package	Uncore W-box perfmon counter MSR.
C94H	3140	MSR_W_PMON_EVNT_SEL2	Package	Uncore W-box perfmon event select MSR.
C95H	3141	MSR_W_PMON_CTR2	Package	Uncore W-box perfmon counter MSR.
C96H	3142	MSR_W_PMON_EVNT_SEL3	Package	Uncore W-box perfmon event select MSR.
C97H	3143	MSR_W_PMON_CTR3	Package	Uncore W-box perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CA0H	3232	MSR_M0_PMON_BOX_CTRL	Package	Uncore M-box 0 perfmon local box control MSR.
CA1H	3233	MSR_M0_PMON_BOX_STATUS	Package	Uncore M-box 0 perfmon local box status MSR.
CA2H	3234	MSR_M0_PMON_BOX_OVF_CTRL	Package	Uncore M-box 0 perfmon local box overflow control MSR.
CA4H	3236	MSR_M0_PMON_TIMESTAMP	Package	Uncore M-box 0 perfmon time stamp unit select MSR.
CA5H	3237	MSR_M0_PMON_DSP	Package	Uncore M-box 0 perfmon DSP unit select MSR.
CA6H	3238	MSR_M0_PMON_ISS	Package	Uncore M-box 0 perfmon ISS unit select MSR.
CA7H	3239	MSR_M0_PMON_MAP	Package	Uncore M-box 0 perfmon MAP unit select MSR.
CA8H	3240	MSR_M0_PMON_MSC_THR	Package	Uncore M-box 0 perfmon MIC THR select MSR.
CA9H	3241	MSR_M0_PMON_PGT	Package	Uncore M-box 0 perfmon PGT unit select MSR.
CAAH	3242	MSR_M0_PMON_PLD	Package	Uncore M-box 0 perfmon PLD unit select MSR.
CABH	3243	MSR_M0_PMON_ZDP	Package	Uncore M-box 0 perfmon ZDP unit select MSR.
CB0H	3248	MSR_M0_PMON_EVNT_SEL0	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTR0	Package	Uncore M-box 0 perfmon counter MSR.
CB2H	3250	MSR_M0_PMON_EVNT_SEL1	Package	Uncore M-box 0 perfmon event select MSR.
CB3H	3251	MSR_M0_PMON_CTR1	Package	Uncore M-box 0 perfmon counter MSR.
CB4H	3252	MSR_M0_PMON_EVNT_SEL2	Package	Uncore M-box 0 perfmon event select MSR.
CB5H	3253	MSR_M0_PMON_CTR2	Package	Uncore M-box 0 perfmon counter MSR.
CB6H	3254	MSR_M0_PMON_EVNT_SEL3	Package	Uncore M-box 0 perfmon event select MSR.
CB7H	3255	MSR_M0_PMON_CTR3	Package	Uncore M-box 0 perfmon counter MSR.
CB8H	3256	MSR_M0_PMON_EVNT_SEL4	Package	Uncore M-box 0 perfmon event select MSR.
CB9H	3257	MSR_M0_PMON_CTR4	Package	Uncore M-box 0 perfmon counter MSR.
CBAH	3258	MSR_M0_PMON_EVNT_SEL5	Package	Uncore M-box 0 perfmon event select MSR.
CBBH	3259	MSR_M0_PMON_CTR5	Package	Uncore M-box 0 perfmon counter MSR.
CC0H	3264	MSR_S1_PMON_BOX_CTRL	Package	Uncore S-box 1 perfmon local box control MSR.
CC1H	3265	MSR_S1_PMON_BOX_STATUS	Package	Uncore S-box 1 perfmon local box status MSR.
CC2H	3266	MSR_S1_PMON_BOX_OVF_CTRL	Package	Uncore S-box 1 perfmon local box overflow control MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CD0H	3280	MSR_S1_PMON_EVNT_SEL0	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTR0	Package	Uncore S-box 1 perfmon counter MSR.
CD2H	3282	MSR_S1_PMON_EVNT_SEL1	Package	Uncore S-box 1 perfmon event select MSR.
CD3H	3283	MSR_S1_PMON_CTR1	Package	Uncore S-box 1 perfmon counter MSR.
CD4H	3284	MSR_S1_PMON_EVNT_SEL2	Package	Uncore S-box 1 perfmon event select MSR.
CD5H	3285	MSR_S1_PMON_CTR2	Package	Uncore S-box 1 perfmon counter MSR.
CD6H	3286	MSR_S1_PMON_EVNT_SEL3	Package	Uncore S-box 1 perfmon event select MSR.
CD7H	3287	MSR_S1_PMON_CTR3	Package	Uncore S-box 1 perfmon counter MSR.
CE0H	3296	MSR_M1_PMON_BOX_CTRL	Package	Uncore M-box 1 perfmon local box control MSR.
CE1H	3297	MSR_M1_PMON_BOX_STATUS	Package	Uncore M-box 1 perfmon local box status MSR.
CE2H	3298	MSR_M1_PMON_BOX_OVF_CTRL	Package	Uncore M-box 1 perfmon local box overflow control MSR.
CE4H	3300	MSR_M1_PMON_TIMESTAMP	Package	Uncore M-box 1 perfmon time stamp unit select MSR.
CE5H	3301	MSR_M1_PMON_DSP	Package	Uncore M-box 1 perfmon DSP unit select MSR.
CE6H	3302	MSR_M1_PMON_ISS	Package	Uncore M-box 1 perfmon ISS unit select MSR.
CE7H	3303	MSR_M1_PMON_MAP	Package	Uncore M-box 1 perfmon MAP unit select MSR.
CE8H	3304	MSR_M1_PMON_MSC_THR	Package	Uncore M-box 1 perfmon MIC THR select MSR.
CE9H	3305	MSR_M1_PMON_PGT	Package	Uncore M-box 1 perfmon PGT unit select MSR.
CEAH	3306	MSR_M1_PMON_PLD	Package	Uncore M-box 1 perfmon PLD unit select MSR.
CEBH	3307	MSR_M1_PMON_ZDP	Package	Uncore M-box 1 perfmon ZDP unit select MSR.
CF0H	3312	MSR_M1_PMON_EVNT_SEL0	Package	Uncore M-box 1 perfmon event select MSR.
CF1H	3313	MSR_M1_PMON_CTR0	Package	Uncore M-box 1 perfmon counter MSR.
CF2H	3314	MSR_M1_PMON_EVNT_SEL1	Package	Uncore M-box 1 perfmon event select MSR.
CF3H	3315	MSR_M1_PMON_CTR1	Package	Uncore M-box 1 perfmon counter MSR.
CF4H	3316	MSR_M1_PMON_EVNT_SEL2	Package	Uncore M-box 1 perfmon event select MSR.
CF5H	3317	MSR_M1_PMON_CTR2	Package	Uncore M-box 1 perfmon counter MSR.
CF6H	3318	MSR_M1_PMON_EVNT_SEL3	Package	Uncore M-box 1 perfmon event select MSR.
CF7H	3319	MSR_M1_PMON_CTR3	Package	Uncore M-box 1 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CF8H	3320	MSR_M1_PMON_EVNT_SEL4	Package	Uncore M-box 1 perfmon event select MSR.
CF9H	3321	MSR_M1_PMON_CTR4	Package	Uncore M-box 1 perfmon counter MSR.
CFAH	3322	MSR_M1_PMON_EVNT_SEL5	Package	Uncore M-box 1 perfmon event select MSR.
CFBH	3323	MSR_M1_PMON_CTR5	Package	Uncore M-box 1 perfmon counter MSR.
D00H	3328	MSR_C0_PMON_BOX_CTRL	Package	Uncore C-box 0 perfmon local box control MSR.
D01H	3329	MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon local box status MSR.
D02H	3330	MSR_C0_PMON_BOX_OVF_CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_C0_PMON_EVNT_SEL0	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_C0_PMON_EVNT_SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_C0_PMON_EVNT_SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_C0_PMON_EVNT_SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_C0_PMON_EVNT_SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_C0_PMON_CTR4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_C0_PMON_EVNT_SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_C0_PMON_CTR5	Package	Uncore C-box 0 perfmon counter MSR.
D20H	3360	MSR_C4_PMON_BOX_CTRL	Package	Uncore C-box 4 perfmon local box control MSR.
D21H	3361	MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon local box status MSR.
D22H	3362	MSR_C4_PMON_BOX_OVF_CTRL	Package	Uncore C-box 4 perfmon local box overflow control MSR.
D30H	3376	MSR_C4_PMON_EVNT_SEL0	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D32H	3378	MSR_C4_PMON_EVNT_SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter MSR.
D34H	3380	MSR_C4_PMON_EVNT_SEL2	Package	Uncore C-box 4 perfmon event select MSR.
D35H	3381	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter MSR.
D36H	3382	MSR_C4_PMON_EVNT_SEL3	Package	Uncore C-box 4 perfmon event select MSR.
D37H	3383	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter MSR.
D38H	3384	MSR_C4_PMON_EVNT_SEL4	Package	Uncore C-box 4 perfmon event select MSR.
D39H	3385	MSR_C4_PMON_CTR4	Package	Uncore C-box 4 perfmon counter MSR.
D3AH	3386	MSR_C4_PMON_EVNT_SEL5	Package	Uncore C-box 4 perfmon event select MSR.
D3BH	3387	MSR_C4_PMON_CTR5	Package	Uncore C-box 4 perfmon counter MSR.
D40H	3392	MSR_C2_PMON_BOX_CTRL	Package	Uncore C-box 2 perfmon local box control MSR.
D41H	3393	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon local box status MSR.
D42H	3394	MSR_C2_PMON_BOX_OVF_CTRL	Package	Uncore C-box 2 perfmon local box overflow control MSR.
D50H	3408	MSR_C2_PMON_EVNT_SEL0	Package	Uncore C-box 2 perfmon event select MSR.
D51H	3409	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter MSR.
D52H	3410	MSR_C2_PMON_EVNT_SEL1	Package	Uncore C-box 2 perfmon event select MSR.
D53H	3411	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter MSR.
D54H	3412	MSR_C2_PMON_EVNT_SEL2	Package	Uncore C-box 2 perfmon event select MSR.
D55H	3413	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter MSR.
D56H	3414	MSR_C2_PMON_EVNT_SEL3	Package	Uncore C-box 2 perfmon event select MSR.
D57H	3415	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter MSR.
D58H	3416	MSR_C2_PMON_EVNT_SEL4	Package	Uncore C-box 2 perfmon event select MSR.
D59H	3417	MSR_C2_PMON_CTR4	Package	Uncore C-box 2 perfmon counter MSR.
D5AH	3418	MSR_C2_PMON_EVNT_SEL5	Package	Uncore C-box 2 perfmon event select MSR.
D5BH	3419	MSR_C2_PMON_CTR5	Package	Uncore C-box 2 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D60H	3424	MSR_C6_PMON_BOX_CTRL	Package	Uncore C-box 6 perfmon local box control MSR.
D61H	3425	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon local box status MSR.
D62H	3426	MSR_C6_PMON_BOX_OVF_CTRL	Package	Uncore C-box 6 perfmon local box overflow control MSR.
D70H	3440	MSR_C6_PMON_EVNT_SEL0	Package	Uncore C-box 6 perfmon event select MSR.
D71H	3441	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter MSR.
D72H	3442	MSR_C6_PMON_EVNT_SEL1	Package	Uncore C-box 6 perfmon event select MSR.
D73H	3443	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter MSR.
D74H	3444	MSR_C6_PMON_EVNT_SEL2	Package	Uncore C-box 6 perfmon event select MSR.
D75H	3445	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter MSR.
D76H	3446	MSR_C6_PMON_EVNT_SEL3	Package	Uncore C-box 6 perfmon event select MSR.
D77H	3447	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter MSR.
D78H	3448	MSR_C6_PMON_EVNT_SEL4	Package	Uncore C-box 6 perfmon event select MSR.
D79H	3449	MSR_C6_PMON_CTR4	Package	Uncore C-box 6 perfmon counter MSR.
D7AH	3450	MSR_C6_PMON_EVNT_SEL5	Package	Uncore C-box 6 perfmon event select MSR.
D7BH	3451	MSR_C6_PMON_CTR5	Package	Uncore C-box 6 perfmon counter MSR.
D80H	3456	MSR_C1_PMON_BOX_CTRL	Package	Uncore C-box 1 perfmon local box control MSR.
D81H	3457	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon local box status MSR.
D82H	3458	MSR_C1_PMON_BOX_OVF_CTRL	Package	Uncore C-box 1 perfmon local box overflow control MSR.
D90H	3472	MSR_C1_PMON_EVNT_SEL0	Package	Uncore C-box 1 perfmon event select MSR.
D91H	3473	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter MSR.
D92H	3474	MSR_C1_PMON_EVNT_SEL1	Package	Uncore C-box 1 perfmon event select MSR.
D93H	3475	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter MSR.
D94H	3476	MSR_C1_PMON_EVNT_SEL2	Package	Uncore C-box 1 perfmon event select MSR.
D95H	3477	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D96H	3478	MSR_C1_PMON_EVNT_SEL3	Package	Uncore C-box 1 perfmon event select MSR.
D97H	3479	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter MSR.
D98H	3480	MSR_C1_PMON_EVNT_SEL4	Package	Uncore C-box 1 perfmon event select MSR.
D99H	3481	MSR_C1_PMON_CTR4	Package	Uncore C-box 1 perfmon counter MSR.
D9AH	3482	MSR_C1_PMON_EVNT_SEL5	Package	Uncore C-box 1 perfmon event select MSR.
D9BH	3483	MSR_C1_PMON_CTR5	Package	Uncore C-box 1 perfmon counter MSR.
DA0H	3488	MSR_C5_PMON_BOX_CTRL	Package	Uncore C-box 5 perfmon local box control MSR.
DA1H	3489	MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon local box status MSR.
DA2H	3490	MSR_C5_PMON_BOX_OVF_CTRL	Package	Uncore C-box 5 perfmon local box overflow control MSR.
DB0H	3504	MSR_C5_PMON_EVNT_SEL0	Package	Uncore C-box 5 perfmon event select MSR.
DB1H	3505	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter MSR.
DB2H	3506	MSR_C5_PMON_EVNT_SEL1	Package	Uncore C-box 5 perfmon event select MSR.
DB3H	3507	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter MSR.
DB4H	3508	MSR_C5_PMON_EVNT_SEL2	Package	Uncore C-box 5 perfmon event select MSR.
DB5H	3509	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter MSR.
DB6H	3510	MSR_C5_PMON_EVNT_SEL3	Package	Uncore C-box 5 perfmon event select MSR.
DB7H	3511	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter MSR.
DB8H	3512	MSR_C5_PMON_EVNT_SEL4	Package	Uncore C-box 5 perfmon event select MSR.
DB9H	3513	MSR_C5_PMON_CTR4	Package	Uncore C-box 5 perfmon counter MSR.
DBAH	3514	MSR_C5_PMON_EVNT_SEL5	Package	Uncore C-box 5 perfmon event select MSR.
DBBH	3515	MSR_C5_PMON_CTR5	Package	Uncore C-box 5 perfmon counter MSR.
DC0H	3520	MSR_C3_PMON_BOX_CTRL	Package	Uncore C-box 3 perfmon local box control MSR.
DC1H	3521	MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon local box status MSR.
DC2H	3522	MSR_C3_PMON_BOX_OVF_CTRL	Package	Uncore C-box 3 perfmon local box overflow control MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD0H	3536	MSR_C3_PMON_EVNT_SEL0	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter MSR.
DD2H	3538	MSR_C3_PMON_EVNT_SEL1	Package	Uncore C-box 3 perfmon event select MSR.
DD3H	3539	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter MSR.
DD4H	3540	MSR_C3_PMON_EVNT_SEL2	Package	Uncore C-box 3 perfmon event select MSR.
DD5H	3541	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter MSR.
DD6H	3542	MSR_C3_PMON_EVNT_SEL3	Package	Uncore C-box 3 perfmon event select MSR.
DD7H	3543	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter MSR.
DD8H	3544	MSR_C3_PMON_EVNT_SEL4	Package	Uncore C-box 3 perfmon event select MSR.
DD9H	3545	MSR_C3_PMON_CTR4	Package	Uncore C-box 3 perfmon counter MSR.
DDAH	3546	MSR_C3_PMON_EVNT_SEL5	Package	Uncore C-box 3 perfmon event select MSR.
DDBH	3547	MSR_C3_PMON_CTR5	Package	Uncore C-box 3 perfmon counter MSR.
DE0H	3552	MSR_C7_PMON_BOX_CTRL	Package	Uncore C-box 7 perfmon local box control MSR.
DE1H	3553	MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon local box status MSR.
DE2H	3554	MSR_C7_PMON_BOX_OVF_CTRL	Package	Uncore C-box 7 perfmon local box overflow control MSR.
DF0H	3568	MSR_C7_PMON_EVNT_SEL0	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter MSR.
DF2H	3570	MSR_C7_PMON_EVNT_SEL1	Package	Uncore C-box 7 perfmon event select MSR.
DF3H	3571	MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter MSR.
DF4H	3572	MSR_C7_PMON_EVNT_SEL2	Package	Uncore C-box 7 perfmon event select MSR.
DF5H	3573	MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter MSR.
DF6H	3574	MSR_C7_PMON_EVNT_SEL3	Package	Uncore C-box 7 perfmon event select MSR.
DF7H	3575	MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter MSR.
DF8H	3576	MSR_C7_PMON_EVNT_SEL4	Package	Uncore C-box 7 perfmon event select MSR.
DF9H	3577	MSR_C7_PMON_CTR4	Package	Uncore C-box 7 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DFAH	3578	MSR_C7_PMON_EVNT_SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTR5	Package	Uncore C-box 7 perfmon counter MSR.
E00H	3584	MSR_R0_PMON_BOX_CTRL	Package	Uncore R-box 0 perfmon local box control MSR.
E01H	3585	MSR_R0_PMON_BOX_STATUS	Package	Uncore R-box 0 perfmon local box status MSR.
E02H	3586	MSR_R0_PMON_BOX_OVF_CTRL	Package	Uncore R-box 0 perfmon local box overflow control MSR.
E04H	3588	MSR_R0_PMON_IPERF0_P0	Package	Uncore R-box 0 perfmon IPERF0 unit Port 0 select MSR.
E05H	3589	MSR_R0_PMON_IPERF0_P1	Package	Uncore R-box 0 perfmon IPERF0 unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERF0_P2	Package	Uncore R-box 0 perfmon IPERF0 unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERF0_P3	Package	Uncore R-box 0 perfmon IPERF0 unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERF0_P4	Package	Uncore R-box 0 perfmon IPERF0 unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERF0_P5	Package	Uncore R-box 0 perfmon IPERF0 unit Port 5 select MSR.
E0AH	3594	MSR_R0_PMON_IPERF0_P6	Package	Uncore R-box 0 perfmon IPERF0 unit Port 6 select MSR.
E0BH	3595	MSR_R0_PMON_IPERF0_P7	Package	Uncore R-box 0 perfmon IPERF0 unit Port 7 select MSR.
E0CH	3596	MSR_R0_PMON_QLX_P0	Package	Uncore R-box 0 perfmon QLX unit Port 0 select MSR.
E0DH	3597	MSR_R0_PMON_QLX_P1	Package	Uncore R-box 0 perfmon QLX unit Port 1 select MSR.
E0EH	3598	MSR_R0_PMON_QLX_P2	Package	Uncore R-box 0 perfmon QLX unit Port 2 select MSR.
E0FH	3599	MSR_R0_PMON_QLX_P3	Package	Uncore R-box 0 perfmon QLX unit Port 3 select MSR.
E10H	3600	MSR_R0_PMON_EVNT_SEL0	Package	Uncore R-box 0 perfmon event select MSR.
E11H	3601	MSR_R0_PMON_CTR0	Package	Uncore R-box 0 perfmon counter MSR.
E12H	3602	MSR_R0_PMON_EVNT_SEL1	Package	Uncore R-box 0 perfmon event select MSR.
E13H	3603	MSR_R0_PMON_CTR1	Package	Uncore R-box 0 perfmon counter MSR.
E14H	3604	MSR_R0_PMON_EVNT_SEL2	Package	Uncore R-box 0 perfmon event select MSR.
E15H	3605	MSR_R0_PMON_CTR2	Package	Uncore R-box 0 perfmon counter MSR.
E16H	3606	MSR_R0_PMON_EVNT_SEL3	Package	Uncore R-box 0 perfmon event select MSR.
E17H	3607	MSR_R0_PMON_CTR3	Package	Uncore R-box 0 perfmon counter MSR.
E18H	3608	MSR_R0_PMON_EVNT_SEL4	Package	Uncore R-box 0 perfmon event select MSR.
E19H	3609	MSR_R0_PMON_CTR4	Package	Uncore R-box 0 perfmon counter MSR.
E1AH	3610	MSR_R0_PMON_EVNT_SEL5	Package	Uncore R-box 0 perfmon event select MSR.
E1BH	3611	MSR_R0_PMON_CTR5	Package	Uncore R-box 0 perfmon counter MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E1CH	3612	MSR_R0_PMON_EVNT_SEL6	Package	Uncore R-box 0 perfmon event select MSR.
E1DH	3613	MSR_R0_PMON_CTR6	Package	Uncore R-box 0 perfmon counter MSR.
E1EH	3614	MSR_R0_PMON_EVNT_SEL7	Package	Uncore R-box 0 perfmon event select MSR.
E1FH	3615	MSR_R0_PMON_CTR7	Package	Uncore R-box 0 perfmon counter MSR.
E20H	3616	MSR_R1_PMON_BOX_CTRL	Package	Uncore R-box 1 perfmon local box control MSR.
E21H	3617	MSR_R1_PMON_BOX_STATUS	Package	Uncore R-box 1 perfmon local box status MSR.
E22H	3618	MSR_R1_PMON_BOX_OVF_CTRL	Package	Uncore R-box 1 perfmon local box overflow control MSR.
E24H	3620	MSR_R1_PMON_IPERF1_P8	Package	Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.
E25H	3621	MSR_R1_PMON_IPERF1_P9	Package	Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.
E26H	3622	MSR_R1_PMON_IPERF1_P10	Package	Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.
E27H	3623	MSR_R1_PMON_IPERF1_P11	Package	Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.
E28H	3624	MSR_R1_PMON_IPERF1_P12	Package	Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.
E29H	3625	MSR_R1_PMON_IPERF1_P13	Package	Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.
E2AH	3626	MSR_R1_PMON_IPERF1_P14	Package	Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.
E2BH	3627	MSR_R1_PMON_IPERF1_P15	Package	Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.
E2CH	3628	MSR_R1_PMON_QLX_P4	Package	Uncore R-box 1 perfmon QLX unit Port 4 select MSR.
E2DH	3629	MSR_R1_PMON_QLX_P5	Package	Uncore R-box 1 perfmon QLX unit Port 5 select MSR.
E2EH	3630	MSR_R1_PMON_QLX_P6	Package	Uncore R-box 1 perfmon QLX unit Port 6 select MSR.
E2FH	3631	MSR_R1_PMON_QLX_P7	Package	Uncore R-box 1 perfmon QLX unit Port 7 select MSR.
E30H	3632	MSR_R1_PMON_EVNT_SEL8	Package	Uncore R-box 1 perfmon event select MSR.
E31H	3633	MSR_R1_PMON_CTR8	Package	Uncore R-box 1 perfmon counter MSR.
E32H	3634	MSR_R1_PMON_EVNT_SEL9	Package	Uncore R-box 1 perfmon event select MSR.
E33H	3635	MSR_R1_PMON_CTR9	Package	Uncore R-box 1 perfmon counter MSR.
E34H	3636	MSR_R1_PMON_EVNT_SEL10	Package	Uncore R-box 1 perfmon event select MSR.
E35H	3637	MSR_R1_PMON_CTR10	Package	Uncore R-box 1 perfmon counter MSR.
E36H	3638	MSR_R1_PMON_EVNT_SEL11	Package	Uncore R-box 1 perfmon event select MSR.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E37H	3639	MSR_R1_PMON_CTR11	Package	Uncore R-box 1 perfmon counter MSR.
E38H	3640	MSR_R1_PMON_EVNT_SEL12	Package	Uncore R-box 1 perfmon event select MSR.
E39H	3641	MSR_R1_PMON_CTR12	Package	Uncore R-box 1 perfmon counter MSR.
E3AH	3642	MSR_R1_PMON_EVNT_SEL13	Package	Uncore R-box 1 perfmon event select MSR.
E3BH	3643	MSR_R1_PMON_CTR13	Package	Uncore R-box 1perfmon counter MSR.
E3CH	3644	MSR_R1_PMON_EVNT_SEL14	Package	Uncore R-box 1 perfmon event select MSR.
E3DH	3645	MSR_R1_PMON_CTR14	Package	Uncore R-box 1 perfmon counter MSR.
E3EH	3646	MSR_R1_PMON_EVNT_SEL15	Package	Uncore R-box 1 perfmon event select MSR.
E3FH	3647	MSR_R1_PMON_CTR15	Package	Uncore R-box 1 perfmon counter MSR.
E45H	3653	MSR_B0_PMON_MATCH	Package	Uncore B-box 0 perfmon local box match MSR.
E46H	3654	MSR_B0_PMON_MASK	Package	Uncore B-box 0 perfmon local box mask MSR.
E49H	3657	MSR_S0_PMON_MATCH	Package	Uncore S-box 0 perfmon local box match MSR.
E4AH	3658	MSR_S0_PMON_MASK	Package	Uncore S-box 0 perfmon local box mask MSR.
E4DH	3661	MSR_B1_PMON_MATCH	Package	Uncore B-box 1 perfmon local box match MSR.
E4EH	3662	MSR_B1_PMON_MASK	Package	Uncore B-box 1 perfmon local box mask MSR.
E54H	3668	MSR_M0_PMON_MM_CONFIG	Package	Uncore M-box 0 perfmon local box address match/mask config MSR.
E55H	3669	MSR_M0_PMON_ADDR_MATCH	Package	Uncore M-box 0 perfmon local box address match MSR.
E56H	3670	MSR_M0_PMON_ADDR_MASK	Package	Uncore M-box 0 perfmon local box address mask MSR.
E59H	3673	MSR_S1_PMON_MATCH	Package	Uncore S-box 1 perfmon local box match MSR.
E5AH	3674	MSR_S1_PMON_MASK	Package	Uncore S-box 1 perfmon local box mask MSR.
E5CH	3676	MSR_M1_PMON_MM_CONFIG	Package	Uncore M-box 1 perfmon local box address match/mask config MSR.
E5DH	3677	MSR_M1_PMON_ADDR_MATCH	Package	Uncore M-box 1 perfmon local box address match MSR.
E5EH	3678	MSR_M1_PMON_ADDR_MASK	Package	Uncore M-box 1 perfmon local box address mask MSR.
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.8.2.2, “Uncore Performance Event Configuration Facility.”

2.7 MSRS IN THE INTEL® XEON® PROCESSOR 5600 SERIES (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor 5600 Series (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-13, Table 2-14, plus additional MSR listed in Table 2-16. These MSRs apply to Intel Core i7, i5 and i3 processor family with CPUID signature DisplayFamily_DisplayModel of 06_25H and 06_2CH, see Table 2-1.

Table 2-16. Additional MSRs Supported by Intel Processors (Based on Intel® Microarchitecture Code Name Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		63:48		Reserved.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.

2.8 MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor E7 Family (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-13 (except MSR address 1ADH), Table 2-14, plus additional MSR listed in Table 2-17. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2FH.

Table 2-17. Additional MSRs Supported by Intel® Xeon® Processor E7 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR.
F41H	3905	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon local box status MSR.
F42H	3906	MSR_C8_PMON_BOX_OVF_CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR.
F50H	3920	MSR_C8_PMON_EVNT_SEL0	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.

Table 2-17. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
F58H	3928	MSR_C8_PMON_EVNT_SEL4	Package	Uncore C-box 8 perfmon event select MSR.
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR.
F5AH	3930	MSR_C8_PMON_EVNT_SEL5	Package	Uncore C-box 8 perfmon event select MSR.
F5BH	3931	MSR_C8_PMON_CTR5	Package	Uncore C-box 8 perfmon counter MSR.
FC0H	4032	MSR_C9_PMON_BOX_CTRL	Package	Uncore C-box 9 perfmon local box control MSR.
FC1H	4033	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon local box status MSR.
FC2H	4034	MSR_C9_PMON_BOX_OVF_CTRL	Package	Uncore C-box 9 perfmon local box overflow control MSR.
FD0H	4048	MSR_C9_PMON_EVNT_SEL0	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter MSR.
FD2H	4050	MSR_C9_PMON_EVNT_SEL1	Package	Uncore C-box 9 perfmon event select MSR.
FD3H	4051	MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter MSR.
FD4H	4052	MSR_C9_PMON_EVNT_SEL2	Package	Uncore C-box 9 perfmon event select MSR.
FD5H	4053	MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter MSR.
FD6H	4054	MSR_C9_PMON_EVNT_SEL3	Package	Uncore C-box 9 perfmon event select MSR.
FD7H	4055	MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter MSR.
FD8H	4056	MSR_C9_PMON_EVNT_SEL4	Package	Uncore C-box 9 perfmon event select MSR.
FD9H	4057	MSR_C9_PMON_CTR4	Package	Uncore C-box 9 perfmon counter MSR.
FDAH	4058	MSR_C9_PMON_EVNT_SEL5	Package	Uncore C-box 9 perfmon event select MSR.
FDBH	4059	MSR_C9_PMON_CTR5	Package	Uncore C-box 9 perfmon counter MSR.

2.9 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 2-18 lists model-specific registers (MSRs) that are common to Intel® processor family based on Intel micro-architecture code name Sandy Bridge. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, 06_2DH, see Table 2-1. Additional MSRs specific to 06_2AH are listed in Table 2-19.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_ SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.
10H	16	IA32_TIME_STAMP_ COUNTER	Thread	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Count SMIs.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Thread	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Thread	**Performance Counter Register** See Table 2-2.
C3H	195	IA32_PMC2	Thread	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Thread	**Performance Counter Register** See Table 2-2.
C5H	197	IA32_PMC4	Core	**Performance Counter Register (if core not shared by threads)**
C6H	198	IA32_PMC5	Core	**Performance Counter Register (if core not shared by threads)**
C7H	199	IA32_PMC6	Core	**Performance Counter Register (if core not shared by threads)**
C8H	200	IA32_PMC7	Core	**Performance Counter Register (if core not shared by threads)**

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_ PERFEVTSEL0	Thread	See Table 2-2.
187H	391	IA32_ PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_ PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_ PERFEVTSEL3	Thread	See Table 2-2.
18AH	394	IA32_ PERFEVTSEL4	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18BH	395	IA32_ PERFEVTSEL5	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18CH	396	IA32_ PERFEVTSEL6	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
18DH	397	IA32_ PERFEVTSEL7	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
198H	408	MSR_PERF_STATUS	Package	
		47:32		Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2^13).
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_ MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2 IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		3:0		**On demand Clock Modulation Duty Cycle (R/W)** In 6.25% increment
		4		**On demand Clock Modulation Enable (R/W)**
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		15:12		Reserved.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable** See Table 2-2
		6:1		Reserved.
		7	Thread	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Thread	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Thread	**xTPR Message Disable (R/W)** See Table 2-2.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33:24		Reserved.
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Unique	
		15:0		Reserved.
		23:16		**Temperature Target (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1A7H	422	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		See http://biosbits.org.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)** See Section 17.8.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
		0		**LBR: Last Branch Record**
		1		**BTF**
		5:2		Reserved.
		6		**TR: Branch Trace**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		63:15		Reserved.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	See http://biosbits.org.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	Always 0 (CMCI not supported).
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		11:8		PEBS_REC_FORMAT. See Table 2-2.
		12		SMM_FREEZE. See Table 2-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS		See Table 2-2. See Section 18.4.2, "Global Counter Control Facilities."
		0	Thread	**Ovf_PMC0**
		1	Thread	**Ovf_PMC1**
		2	Thread	**Ovf_PMC2**

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3	Thread	**Ovf_PMC3**
		4	Core	**Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Core	**Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		60:35		Reserved.
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
		0	Thread	**Set 1 to enable PMC0 to count**
		1	Thread	**Set 1 to enable PMC1 to count**
		2	Thread	**Set 1 to enable PMC2 to count**
		3	Thread	**Set 1 to enable PMC3 to count**
		4	Core	**Set 1 to enable PMC4 to count (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Set 1 to enable PMC5 to count (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Set 1 to enable PMC6 to count (if CPUID.0AH:EAX[15:8] > 6)**
		7	Core	**Set 1 to enable PMC7 to count (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to enable FixedCtr0 to count**
		33	Thread	**Set 1 to enable FixedCtr1 to count**
		34	Thread	**Set 1 to enable FixedCtr2 to count**
		63:35		Reserved.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL		See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
		0	Thread	**Set 1 to clear Ovf_PMC0**
		1	Thread	**Set 1 to clear Ovf_PMC1**
		2	Thread	**Set 1 to clear Ovf_PMC2**
		3	Thread	**Set 1 to clear Ovf_PMC3**
		4	Core	**Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		7	Core	**Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to clear Ovf_FixedCtr0**
		33	Thread	**Set 1 to clear Ovf_FixedCtr1**
		34	Thread	**Set 1 to clear Ovf_FixedCtr2**
		60:35		Reserved.
		61	Thread	**Set 1 to clear Ovf_Uncore**
		62	Thread	**Set 1 to clear Ovf_BufDSSAVE**
		63	Thread	**Set 1 to clear CondChgd**
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.8.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		62:36		Reserved.
		63		Enable Precise Store. (R/W)
3F6H	1014	MSR_PEBS_LD_LAT	Thread	see See Section 18.8.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		**PCU Hardware Error (R/W)** When set, enables signaling of PCU hardware detected errors.
		1		**PCU Controller Error (R/W)** When set, enables signaling of PCU controller detected errors
		2		**PCU Firmware Error (R/W)** When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, “VMCS Enumeration.”
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, “VM-Execution Controls.”
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Thread	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
4C3H	1219	IA32_A_PMC2	Thread	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Thread	See Table 2-2.
4C5H	1221	IA32_A_PMC4	Core	See Table 2-2.
4C6H	1222	IA32_A_PMC5	Core	See Table 2-2.
4C7H	1223	IA32_A_PMC6	Core	See Table 2-2.
4C8H	1224	IA32_A_PMC7	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, “Debug Store (DS) Mechanism.”
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, “RAPL Interfaces.”
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	**Package C6 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		**Package C2 Residency Counter. (R/O)** Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.8.1 and record format in Section 17.4.8.1
681H	1665	MSR_ LASTBRANCH_1_FROM_IP	Thread	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_ LASTBRANCH_2_FROM_IP	Thread	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_ LASTBRANCH_3_FROM_IP	Thread	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_ LASTBRANCH_4_FROM_IP	Thread	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_ LASTBRANCH_5_FROM_IP	Thread	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_ LASTBRANCH_6_FROM_IP	Thread	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Thread	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Thread	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Thread	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Thread	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 2-2.
802H-83FH		X2APIC MSRs	Thread	See Table 2-2.
C000_ 0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature (R/W)** See Table 2-2 and Section 17.16.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.9.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-19 and Table 2-20 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family (based on Intel microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH; see Table 2-1.

Table 2-19. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
60CH	1548	MSR_PKGC7_IRTL	Package	**Package C7 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-19. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
63AH	1594	MSR_PP0_POLICY	Package	**PP0 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-18, Table 2-19, and Table 2-20 for MSR definitions applicable to processors with CPUID signature 06_2AH.				

Table 2-20 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06_2AH.

Table 2-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4 select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation

Table 2-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
702H	1794	MSR_UNC_CBO_0_PERFEVTSEL2	Package	Uncore C-Box 0, counter 2 event select MSR.
703H	1795	MSR_UNC_CBO_0_PERFEVTSEL3	Package	Uncore C-Box 0, counter 3 event select MSR.
705H	1797	MSR_UNC_CBO_0_UNIT_STATUS	Package	Uncore C-Box 0, unit status for counter 0-3
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
708H	1800	MSR_UNC_CBO_0_PERFCTR2	Package	Uncore C-Box 0, performance counter 2.
709H	1801	MSR_UNC_CBO_0_PERFCTR3	Package	Uncore C-Box 0, performance counter 3.
710H	1808	MSR_UNC_CBO_1_PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
712H	1810	MSR_UNC_CBO_1_PERFEVTSEL2	Package	Uncore C-Box 1, counter 2 event select MSR.
713H	1811	MSR_UNC_CBO_1_PERFEVTSEL3	Package	Uncore C-Box 1, counter 3 event select MSR.

Table 2-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
715H	1813	MSR_UNC_CBO_1_UNIT_STATUS	Package	Uncore C-Box 1, unit status for counter 0-3
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
718H	1816	MSR_UNC_CBO_1_PERFCTR2	Package	Uncore C-Box 1, performance counter 2.
719H	1817	MSR_UNC_CBO_1_PERFCTR3	Package	Uncore C-Box 1, performance counter 3.
720H	1824	MSR_UNC_CBO_2_PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
722H	1826	MSR_UNC_CBO_2_PERFEVTSEL2	Package	Uncore C-Box 2, counter 2 event select MSR.
723H	1827	MSR_UNC_CBO_2_PERFEVTSEL3	Package	Uncore C-Box 2, counter 3 event select MSR.
725H	1829	MSR_UNC_CBO_2_UNIT_STATUS	Package	Uncore C-Box 2, unit status for counter 0-3
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
728H	1832	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
729H	1833	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
730H	1840	MSR_UNC_CBO_3_PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
732H	1842	MSR_UNC_CBO_3_PERFEVTSEL2	Package	Uncore C-Box 3, counter 2 event select MSR.
733H	1843	MSR_UNC_CBO_3_PERFEVTSEL3	Package	Uncore C-Box 3, counter 3 event select MSR.
735H	1845	MSR_UNC_CBO_3_UNIT_STATUS	Package	Uncore C-Box 3, unit status for counter 0-3
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
738H	1848	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
739H	1849	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
740H	1856	MSR_UNC_CBO_4_PERFEVTSEL0	Package	Uncore C-Box 4, counter 0 event select MSR
741H	1857	MSR_UNC_CBO_4_PERFEVTSEL1	Package	Uncore C-Box 4, counter 1 event select MSR.
742H	1858	MSR_UNC_CBO_4_PERFEVTSEL2	Package	Uncore C-Box 4, counter 2 event select MSR.

Table 2-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
743H	1859	MSR_UNC_CBO_4_PERFEVTSEL3	Package	Uncore C-Box 4, counter 3 event select MSR.
745H	1861	MSR_UNC_CBO_4_UNIT_STATUS	Package	Uncore C-Box 4, unit status for counter 0-3
746H	1862	MSR_UNC_CBO_4_PERFCTR0	Package	Uncore C-Box 4, performance counter 0.
747H	1863	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, performance counter 1.
748H	1864	MSR_UNC_CBO_4_PERFCTR2	Package	Uncore C-Box 4, performance counter 2.
749H	1865	MSR_UNC_CBO_4_PERFCTR3	Package	Uncore C-Box 4, performance counter 3.

2.9.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-21 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family (based on Intel® microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2DH, and also supports MSRs listed in Table 2-18 and Table 2-22.

Table 2-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.

Table 2-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
39CH	924	MSR_PEBS_NUM_ALT	Package	
		0		**ENABLE_PEBS_NUM_ALT (RW)** Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see Table 19-16
		63:1		Reserved (must be zero).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”

Table 2-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATU S	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-18, Table 2-21, and Table 2-22 for MSR definitions applicable to processors with CPUID signature 06_2DH.				

2.9.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-22. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2DH

Table 2-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C08H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
C09H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

Table 2-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C10H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
C11H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
C16H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
C17H		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
C24H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
C30H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
C31H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
C32H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
C33H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
C34H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
C36H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
C37H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
C38H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
C39H		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
D04H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon local box wide control.
D10H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
D11H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
D12H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
D13H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
D14H		MSR_C0_PMON_BOX_FILTER	Package	Uncore C-box 0 perfmon box wide filter.
D16H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
D17H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
D18H		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
D19H		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
D24H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon local box wide control.
D30H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
D31H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
D32H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
D33H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
D34H		MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 perfmon box wide filter.
D36H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
D37H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
D38H		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
D39H		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
D44H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon local box wide control.
D50H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
D51H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.

Table 2-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D52H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
D53H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
D54H		MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 perfmon box wide filter.
D56H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
D57H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
D58H		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
D59H		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
D64H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon local box wide control.
D70H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
D71H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
D72H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
D73H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
D74H		MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 perfmon box wide filter.
D76H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
D77H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
D78H		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
D79H		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
D84H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon local box wide control.
D90H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
D91H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
D92H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
D93H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
D94H		MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 perfmon box wide filter.
D96H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
D97H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
D98H		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
D99H		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
DA4H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon local box wide control.
DB0H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
DB1H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
DB2H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
DB3H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
DB4H		MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 perfmon box wide filter.
DB6H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
DB7H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
DB8H		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
DB9H		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.

Table 2-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC4H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon local box wide control.
DD0H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
DD1H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
DD2H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
DD3H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
DD4H		MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 perfmon box wide filter.
DD6H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
DD7H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
DD8H		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
DD9H		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
DE4H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon local box wide control.
DF0H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
DF1H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
DF2H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
DF3H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
DF4H		MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 perfmon box wide filter.
DF6H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
DF7H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
DF8H		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
DF9H		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.

2.10 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME IVY BRIDGE)

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family (based on Intel microarchitecture code name Ivy Bridge) support the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-20, and Table 2-23. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3AH.

Table 2-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.

Table 2-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.

Table 2-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.

Table 2-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
See Table 2-18, Table 2-19 and Table 2-20 for other MSR definitions applicable to processors with CPUID signature 06_3AH				

2.10.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 2-24 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3EH, see Table 2-1. These processors supports the MSR interfaces listed in Table 2-18, and Table 2-24.

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		**Protected Processor Inventory Number (R/O)** A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for privileged system inventory agent to read PPIN from MSR_PPIN. When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		Count
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		Reserved.
		26		**MCG_ELOG_P**
		63:27		Reserved.
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		**Temperature Target (RO)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		27:24		**TCC Activation Offset (R/W)** Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT 1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		63:32		Reserved
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
296H	662	IA32_MC22_CTL2	Package	See Table 2-2.
297H	663	IA32_MC23_CTL2	Package	See Table 2-2.
298H	664	IA32_MC24_CTL2	Package	See Table 2-2.
299H	665	IA32_MC25_CTL2	Package	See Table 2-2.
29AH	666	IA32_MC26_CTL2	Package	See Table 2-2.
29BH	667	IA32_MC27_CTL2	Package	See Table 2-2.
29CH	668	IA32_MC28_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC5 reports MC error from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC7 and MC 8 report MC error from the two home agents.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC7 and MC 8 report MC error from the two home agents.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
42DH	1069	IA32_MC11_STATUS	Package	Bank MC11 reports MC error from a specific channel of the integrated memory controller.
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	Bank MC20 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
458H	1112	IA32_MC22_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC22 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
459H	1113	IA32_MC22_STATUS	Package	
45AH	1114	IA32_MC22_ADDR	Package	
45BH	1115	IA32_MC22_MISC	Package	
45CH	1116	IA32_MC23_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC23 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
45DH	1117	IA32_MC23_STATUS	Package	
45EH	1118	IA32_MC23_ADDR	Package	
45FH	1119	IA32_MC23_MISC	Package	
460H	1120	IA32_MC24_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC24 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
461H	1121	IA32_MC24_STATUS	Package	
462H	1122	IA32_MC24_ADDR	Package	
463H	1123	IA32_MC24_MISC	Package	
464H	1124	IA32_MC25_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC25 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
465H	1125	IA32_MC25_STATUS	Package	
466H	1126	IA32_MC25_ADDR	Package	
467H	1127	IA32_MC2MISC	Package	
468H	1128	IA32_MC26_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC26 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
469H	1129	IA32_MC26_STATUS	Package	
46AH	1130	IA32_MC26_ADDR	Package	
46BH	1131	IA32_MC26_MISC	Package	
46CH	1132	IA32_MC27_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC27 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
46DH	1133	IA32_MC27_STATUS	Package	
46EH	1134	IA32_MC27_ADDR	Package	
46FH	1135	IA32_MC27_MISC	Package	

Table 2-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
470H	1136	IA32_MC28_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC28 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
471H	1137	IA32_MC28_STATUS	Package	
472H	1138	IA32_MC28_ADDR	Package	
473H	1139	IA32_MC28_MISC	Package	
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-18, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06_3EH				

2.10.2 Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family

Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with CPUID DisplayFamily_DisplayModel signature 06_3EH supports the MSR interfaces listed in Table 2-18, Table 2-24, and Table 2-25.

Table 2-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		Count
		8		**MCG_CTL_P**

Table 2-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		63:25		Reserved.
17AH	378	IA32_MCG_STATUS	Thread	(R/W0)
		0		**RIPV**
		1		**EIPV**
		2		**MCIP**
		3		**LMCE signaled**
		63:4		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		62:56		Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
29DH	669	IA32_MC29_CTL2	Package	See Table 2-2.
29EH	670	IA32_MC30_CTL2	Package	See Table 2-2.
29FH	671	IA32_MC31_CTL2	Package	See Table 2-2.

Table 2-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.8.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
41BH	1051	IA32_MC6_MISC	Package	Misc MAC information of Integrated I/O. (R/O) see Section 15.3.2.4
		5:0		Recoverable Address LSB
		8:6		Address Mode
		15:9		Reserved
		31:16		PCI Express Requestor ID
		39:32		PCI Express Segment Number
		63:32		Reserved
474H	1140	IA32_MC29_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC29 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
475H	1141	IA32_MC29_STATUS	Package	
476H	1142	IA32_MC29_ADDR	Package	
477H	1143	IA32_MC29_MISC	Package	
478H	1144	IA32_MC30_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC30 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
479H	1145	IA32_MC30_STATUS	Package	
47AH	1146	IA32_MC30_ADDR	Package	
47BH	1147	IA32_MC30_MISC	Package	
47CH	1148	IA32_MC31_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC31 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
47DH	1149	IA32_MC31_STATUS	Package	
47EH	1150	IA32_MC31_ADDR	Package	
47FH	1147	IA32_MC31_MISC	Package	
See Table 2-18, Table 2-24 for other MSR definitions applicable to Intel Xeon processor E7 v2 with CPUID signature 06_3AH.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.10.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-22 and Table 2-26. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3EH.

Table 2-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C00H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
C01H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
C06H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
C15H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
C35H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
D1AH		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter1.
D3AH		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
D5AH		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
D7AH		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
D9AH		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
DBAH		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
DDAH		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
DFAH		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E04H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E10H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E11H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E12H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E13H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E14H		MSR_C8_PMON_BOX_FILTER	Package	Uncore C-box 8 perfmon box wide filter.
E16H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E17H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E18H		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E19H		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E1AH		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E24H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E30H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E31H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E32H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E33H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E34H		MSR_C9_PMON_BOX_FILTER	Package	Uncore C-box 9 perfmon box wide filter.
E36H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E37H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.

Table 2-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E38H		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E39H		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
E3AH		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E44H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
E50H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
E51H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
E52H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
E53H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
E54H		MSR_C10_PMON_BOX_FILTER	Package	Uncore C-box 10 perfmon box wide filter.
E56H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.
E57H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
E58H		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
E59H		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
E5AH		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
E64H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
E70H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
E71H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
E72H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
E73H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
E74H		MSR_C11_PMON_BOX_FILTER	Package	Uncore C-box 11 perfmon box wide filter.
E76H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
E77H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
E78H		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
E79H		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
E7AH		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
E84H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
E90H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
E91H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
E92H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
E93H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
E94H		MSR_C12_PMON_BOX_FILTER	Package	Uncore C-box 12 perfmon box wide filter.
E96H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
E97H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
E98H		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
E99H		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
E9AH		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EA4H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.

Table 2-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB0H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
EB1H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
EB2H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
EB3H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
EB4H		MSR_C13_PMON_BOX_FILTER	Package	Uncore C-box 13 perfmon box wide filter.
EB6H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
EB7H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EB8H		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EB9H		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EBAH		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
EC4H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
ED0H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
ED1H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
ED2H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
ED3H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
ED4H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter.
ED6H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
ED7H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
ED8H		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
ED9H		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EDAH		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.

2.11 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS (BASED ON HASWELL MICROARCHITECTURE)

The 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with CPUID DisplayFamily_DisplayModel signature 06_3CH/06_45H/06_46H, support the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-20, and Table 2-27. For an MSR listed in Table 2-18 that also appears in Table 2-27, Table 2-27 supercede Table 2-18.

The MSRs listed in Table 2-27 also apply to processors based on Haswell-E microarchitecture (see Section 2.12).

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	
		7:0		Reserved.

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
186H	390	IA32_PERFEVTSEL0	THREAD	**Performance Event Select for Counter 0 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.11.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
187H	391	IA32_PERFEVTSEL1	THREAD	**Performance Event Select for Counter 1 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.11.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
188H	392	IA32_PERFEVTSEL2	THREAD	**Performance Event Select for Counter 2 (R/W)** Supports all fields described inTable 2-2 and the fields below.

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		32		IN_TX: see Section 18.11.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
		33		IN_TXCP: see Section 18.11.5.1 When IN_TXCP=1 & IN_TX=1 and in sampling, spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large “sample-after” value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.
189H	393	IA32_PERFEVTSEL3	THREAD	**Performance Event Select for Counter 3 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.11.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)**
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		9		**EN_CALL_STACK**
		63:9		Reserved.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
		0		**LBR: Last Branch Record**
		1		**BTF**
		5:2		Reserved.
		6		**TR: Branch Trace**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		15		**RTM_DEBUG**
		63:15		Reserved.
491H	1169	IA32_VMX_VMFUNC	THREAD	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
60BH	1548	MSR_PKGC_IRTL1	Package	**Package C6/C7 Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-18 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C6/C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-18 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, “Package RAPL Domain.”
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, “DRAM RAPL Domain.”
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, “DRAM RAPL Domain.”
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		62:47		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		62:47		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.

Table 2-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
C80H	3200	IA32_DEBUG_INTERFACE	Package	**Silicon Debug Feature Control (R/W)** See Table 2-2.

2.11.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 2-28 lists model-specific registers (MSRs) that are specific to 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3CH/06_45H/06_46H, see Table 2-1.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processor with signature 06_3CH
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Encoded number of C-Box, derive value by "-1"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W)** **(frequency refers to processor graphics frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Graphics Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:30		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W) (frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		5:2		Reserved.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
700H	1792	MSR_UNC_CBO_0_PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1

Table 2-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
710H	1808	MSR_UNC_CBO_1_PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1824	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
See Table 2-18, Table 2-19, Table 2-20, Table 2-23, Table 2-27 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06_46H.				

2.11.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with CPUID DisplayFamily_DisplayModel signature 06_45H supports the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-27, Table 2-28, and Table 2-29.

Table 2-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		59:0		Package C10 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved
See Table 2-18, Table 2-19, Table 2-20, Table 2-27, Table 2-28 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

2.12 MSRS IN INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

Intel® Xeon® processor E5 v3 family and Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID DisplayFamily_DisplayModel = 06_3F). These processors supports the MSR interfaces listed in Table 2-18, Table 2-27, and Table 2-30.

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	**Configured State of Enabled Processor Core Count and Logical Processor Count (RO)** ▪ After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package. ▪ Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.
		15:0		**Core_COUNT (RO)** The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		31:16		**THREAD_COUNT (RO)** The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		63:32		Reserved
53H	83	MSR_THREAD_ID_INFO	Thread	**A Hardware Assigned ID for the Logical Processor (RO)**
		7:0		**Logical_Processor_ID (RO)** An implementation-specific numerical. value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.
		63:8		Reserved

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		Count
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10 core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		63:56	Package	**Maximum Ratio Limit for16C** Maximum turbo ratio limit of 16 core active.
1AFH	431	MSR_TURBO_RATIO_LIMIT2	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 17C** Maximum turbo ratio limit of 17 core active.
		15:8	Package	**Maximum Ratio Limit for 18C** Maximum turbo ratio limit of 18 core active.
		62:16	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, “RAPL Interfaces.”
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, “RAPL Interfaces.”
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, “DRAM RAPL Domain.”

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** Energy Consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
61EH	1566	MSR_PCIE_PLL_RATIO	Package	**Configuration of PCIE PLL Relative to BCLK(R/W)**
		1:0	Package	**PCIE Ratio (R/W)** 00b: Use 5:5 mapping for100MHz operation (default) 01b: Use 5:4 mapping for125MHz operation 10b: Use 5:3 mapping for166MHz operation 11b: Use 5:2 mapping for250MHz operation
		2	Package	**LPLL Select (R/W)** if 1, use configured setting of PCIE Ratio
		3	Package	**LONG RESET (R/W)** if 1, wait additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Power Budget Management Status (R0)** When set, frequency is reduced below the operating system request due to PBM limit
		3		**Platform Configuration Services Status (R0)** When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		**Multi-Core Turbo Log** When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		**Core Frequency P1 Log** When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Core Max n-core Turbo Frequency Limiting Log** When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		**Core Frequency Limiting Log** When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x0: no monitoring** **0x1: L3 occupancy monitoring** **all other encoding reserved.**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8EH	3214	IA32_QM_CTR	THREAD	**Monitoring Counter Register (R/O).** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		61:0		**Resource Monitored Data**

Table 2-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		62		**Unavailable**: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.
		63		**Error:** If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W).**
		9:0		**RMID**
		63: 10		**Reserved**
See Table 2-18, Table 2-27 for other MSR definitions applicable to processors with CPUID signature 06_3FH.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.12.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

Intel Xeon Processor E5 v3 and E7 v3 family are based on the Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-31. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3FH.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
700H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
701H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
702H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
703H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
704H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter
705H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
706H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
708H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
709H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
70AH		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
710H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
711H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
712H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
713H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
714H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
715H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
716H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
717H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
718H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
719H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
71AH		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
720H		MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 perfmon for SBo 0 box-wide control
721H		MSR_S0_PMON_EVNTSEL0	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 0.
722H		MSR_S0_PMON_EVNTSEL1	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 1.
723H		MSR_S0_PMON_EVNTSEL2	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 2.
724H		MSR_S0_PMON_EVNTSEL3	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 3.
725H		MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 perfmon box-wide filter.
726H		MSR_S0_PMON_CTR0	Package	Uncore SBo 0 perfmon counter 0.
727H		MSR_S0_PMON_CTR1	Package	Uncore SBo 0 perfmon counter 1.
728H		MSR_S0_PMON_CTR2	Package	Uncore SBo 0 perfmon counter 2.
729H		MSR_S0_PMON_CTR3	Package	Uncore SBo 0 perfmon counter 3.
72AH		MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 perfmon for SBo 1 box-wide control
72BH		MSR_S1_PMON_EVNTSEL0	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 0.
72CH		MSR_S1_PMON_EVNTSEL1	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 1.
72DH		MSR_S1_PMON_EVNTSEL2	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 2.
72EH		MSR_S1_PMON_EVNTSEL3	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 3.
72FH		MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 perfmon box-wide filter.
730H		MSR_S1_PMON_CTR0	Package	Uncore SBo 1 perfmon counter 0.
731H		MSR_S1_PMON_CTR1	Package	Uncore SBo 1 perfmon counter 1.
732H		MSR_S1_PMON_CTR2	Package	Uncore SBo 1 perfmon counter 2.
733H		MSR_S1_PMON_CTR3	Package	Uncore SBo 1 perfmon counter 3.
734H		MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 perfmon for SBo 2 box-wide control
735H		MSR_S2_PMON_EVNTSEL0	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 0.
736H		MSR_S2_PMON_EVNTSEL1	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 1.
737H		MSR_S2_PMON_EVNTSEL2	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 2.
738H		MSR_S2_PMON_EVNTSEL3	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 3.
739H		MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 perfmon box-wide filter.
73AH		MSR_S2_PMON_CTR0	Package	Uncore SBo 2 perfmon counter 0.
73BH		MSR_S2_PMON_CTR1	Package	Uncore SBo 2 perfmon counter 1.
73CH		MSR_S2_PMON_CTR2	Package	Uncore SBo 2 perfmon counter 2.
73DH		MSR_S2_PMON_CTR3	Package	Uncore SBo 2 perfmon counter 3.
73EH		MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 perfmon for SBo 3 box-wide control
73FH		MSR_S3_PMON_EVNTSEL0	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 0.
740H		MSR_S3_PMON_EVNTSEL1	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 1.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
741H		MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 2.
742H		MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 3.
743H		MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 perfmon box-wide filter.
744H		MSR_S3_PMON_CTR0	Package	Uncore SBo 3 perfmon counter 0.
745H		MSR_S3_PMON_CTR1	Package	Uncore SBo 3 perfmon counter 1.
746H		MSR_S3_PMON_CTR2	Package	Uncore SBo 3 perfmon counter 2.
747H		MSR_S3_PMON_CTR3	Package	Uncore SBo 3 perfmon counter 3.
E00H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon for box-wide control
E01H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
E02H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
E03H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
E04H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
E05H		MSR_C0_PMON_BOX_FILTER0	Package	Uncore C-box 0 perfmon box wide filter 0.
E06H		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter 1.
E07H		MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon box wide status.
E08H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
E09H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
E0AH		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
E0BH		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
E10H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon for box-wide control
E11H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
E12H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
E13H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
E14H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
E15H		MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-box 1 perfmon box wide filter 0.
E16H		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
E17H		MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon box wide status.
E18H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
E19H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
E1AH		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
E1BH		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
E20H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon for box-wide control
E21H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
E22H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.
E23H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
E24H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
E25H		MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-box 2 perfmon box wide filter 0.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E26H		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
E27H		MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon box wide status.
E28H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
E29H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
E2AH		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
E2BH		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
E30H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon for box-wide control
E31H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
E32H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
E33H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
E34H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
E35H		MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-box 3 perfmon box wide filter 0.
E36H		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
E37H		MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon box wide status.
E38H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
E39H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
E3AH		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
E3BH		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
E40H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon for box-wide control
E41H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
E42H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
E43H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
E44H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
E45H		MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-box 4 perfmon box wide filter 0.
E46H		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
E47H		MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon box wide status.
E48H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
E49H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
E4AH		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
E4BH		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
E50H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon for box-wide control
E51H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
E52H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
E53H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
E54H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
E55H		MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-box 5 perfmon box wide filter 0.
E56H		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E57H		MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon box wide status.
E58H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
E59H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
E5AH		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
E5BH		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.
E60H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon for box-wide control
E61H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
E62H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
E63H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
E64H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
E65H		MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-box 6 perfmon box wide filter 0.
E66H		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
E67H		MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon box wide status.
E68H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
E69H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
E6AH		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
E6BH		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
E70H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon for box-wide control.
E71H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
E72H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
E73H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
E74H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
E75H		MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-box 7 perfmon box wide filter 0.
E76H		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E77H		MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon box wide status.
E78H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
E79H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
E7AH		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
E7BH		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.
E80H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E81H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E82H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E83H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E84H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E85H		MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-box 8 perfmon box wide filter0.
E86H		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E87H		MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon box wide status.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E88H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E89H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E8AH		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E8BH		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E90H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E91H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E92H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E93H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E94H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E95H		MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-box 9 perfmon box wide filter0.
E96H		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E97H		MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon box wide status.
E98H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E99H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.
E9AH		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E9BH		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
EA0H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
EA1H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
EA2H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
EA3H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
EA4H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
EA5H		MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-box 10 perfmon box wide filter0.
EA6H		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
EA7H		MSR_C10_PMON_BOX_STATUS	Package	Uncore C-box 10 perfmon box wide status.
EA8H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.
EA9H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
EAAH		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
EABH		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
EB0H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
EB1H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
EB2H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
EB3H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
EB4H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
EB5H		MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-box 11 perfmon box wide filter0.
EB6H		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
EB7H		MSR_C11_PMON_BOX_STATUS	Package	Uncore C-box 11 perfmon box wide status.
EB8H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB9H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
EBAH		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
EBBH		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
EC0H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
EC1H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
EC2H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
EC3H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
EC4H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
EC5H		MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-box 12 perfmon box wide filter0.
EC6H		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EC7H		MSR_C12_PMON_BOX_STATUS	Package	Uncore C-box 12 perfmon box wide status.
EC8H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
EC9H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
ECAH		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
ECBH		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
ED0H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.
ED1H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
ED2H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
ED3H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
ED4H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
ED5H		MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-box 13 perfmon box wide filter0.
ED6H		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
ED7H		MSR_C13_PMON_BOX_STATUS	Package	Uncore C-box 13 perfmon box wide status.
ED8H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
ED9H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EDAH		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EDBH		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EE0H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
EE1H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
EE2H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
EE3H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
EE4H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
EE5H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter0.
EE6H		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.
EE7H		MSR_C14_PMON_BOX_STATUS	Package	Uncore C-box 14 perfmon box wide status.
EE8H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
EE9H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EEAH		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
EEBH		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EF0H		MSR_C15_PMON_BOX_CTL	Package	Uncore C-box 15 perfmon local box wide control.
EF1H		MSR_C15_PMON_EVNTSEL0	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 0.
EF2H		MSR_C15_PMON_EVNTSEL1	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 1.
EF3H		MSR_C15_PMON_EVNTSEL2	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 2.
EF4H		MSR_C15_PMON_EVNTSEL3	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 3.
EF5H		MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-box 15 perfmon box wide filter0.
EF6H		MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-box 15 perfmon box wide filter1.
EF7H		MSR_C15_PMON_BOX_STATUS	Package	Uncore C-box 15 perfmon box wide status.
EF8H		MSR_C15_PMON_CTR0	Package	Uncore C-box 15 perfmon counter 0.
EF9H		MSR_C15_PMON_CTR1	Package	Uncore C-box 15 perfmon counter 1.
EFAH		MSR_C15_PMON_CTR2	Package	Uncore C-box 15 perfmon counter 2.
EFBH		MSR_C15_PMON_CTR3	Package	Uncore C-box 15 perfmon counter 3.
F00H		MSR_C16_PMON_BOX_CTL	Package	Uncore C-box 16 perfmon for box-wide control
F01H		MSR_C16_PMON_EVNTSEL0	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 0.
F02H		MSR_C16_PMON_EVNTSEL1	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 1.
F03H		MSR_C16_PMON_EVNTSEL2	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 2.
F04H		MSR_C16_PMON_EVNTSEL3	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 3.
F05H		MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-box 16 perfmon box wide filter 0.
F06H		MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-box 16 perfmon box wide filter 1.
F07H		MSR_C16_PMON_BOX_STATUS	Package	Uncore C-box 16 perfmon box wide status.
F08H		MSR_C16_PMON_CTR0	Package	Uncore C-box 16 perfmon counter 0.
F09H		MSR_C16_PMON_CTR1	Package	Uncore C-box 16 perfmon counter 1.
F0AH		MSR_C16_PMON_CTR2	Package	Uncore C-box 16 perfmon counter 2.
E0BH		MSR_C16_PMON_CTR3	Package	Uncore C-box 16 perfmon counter 3.
F10H		MSR_C17_PMON_BOX_CTL	Package	Uncore C-box 17 perfmon for box-wide control
F11H		MSR_C17_PMON_EVNTSEL0	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 0.
F12H		MSR_C17_PMON_EVNTSEL1	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 1.
F13H		MSR_C17_PMON_EVNTSEL2	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 2.
F14H		MSR_C17_PMON_EVNTSEL3	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 3.
F15H		MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-box 17 perfmon box wide filter 0.
F16H		MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-box 17 perfmon box wide filter1.
F17H		MSR_C17_PMON_BOX_STATUS	Package	Uncore C-box 17 perfmon box wide status.
F18H		MSR_C17_PMON_CTR0	Package	Uncore C-box 17 perfmon counter 0.
F19H		MSR_C17_PMON_CTR1	Package	Uncore C-box 17 perfmon counter 1.
F1AH		MSR_C17_PMON_CTR2	Package	Uncore C-box 17 perfmon counter 2.

Table 2-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
F1BH		MSR_C17_PMON_CTR3	Package	Uncore C-box 17 perfmon counter 3.

2.13 MSRS IN INTEL® CORE™ M PROCESSORS AND 5TH GENERATION INTEL CORE PROCESSORS

The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors, and Intel® Xeon® Processor E3-1200 v4 family are based on the Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have CPUID DisplayFamily_DisplayModel signature 06_3DH. Intel® Xeon® Processor E3-1200 v4 family and the 5th generation Intel® Core™ Processors have CPUID DisplayFamily_DisplayModel signature 06_47H. Processors with signatures 06_3DH and 06_47H support the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-20, Table 2-23, Table 2-27, Table 2-28, Table 2-32, and Table 2-33. For an MSR listed in Table 2-33 that also appears in the model-specific tables of prior generations, Table 2-33 supercede prior generation tables.

Table 2-32 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID signatures 06_3DH, 06_47H, 06_4FH, and 06_56H).

Table 2-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
		0		**Ovf_PMC0**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**
		33		**Ovf_FixedCtr1**
		34		**Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Trace_ToPA_PMI. See** Section 35.2.6.2, “Table of Physical Addresses (ToPA).”
		60:56		Reserved.
		61		**Ovf_Uncore**
		62		**Ovf_BufDSSAVE**
		63		**CondChgd**
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.4.2, “Global Counter Control Facilities.”
		0		**Set 1 to clear Ovf_PMC0**
		1		**Set 1 to clear Ovf_PMC1**

Table 2-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**Set 1 to clear Ovf_PMC2**
		3		**Set 1 to clear Ovf_PMC3**
		31:4		Reserved.
		32		**Set 1 to clear Ovf_FixedCtr0**
		33		**Set 1 to clear Ovf_FixedCtr1**
		34		**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Set 1 to clear Trace_ToPA_PMI. See** Section 35.2.6.2, “Table of Physical Addresses (ToPA).”
		60:56		Reserved.
		61		**Set 1 to clear Ovf_Uncore**
		62		**Set 1 to clear Ovf_BufDSSAVE**
		63		**Set 1 to clear CondChgd**
560H	1376	IA32_RTIT_OUTPUT_BASE	THREAD	**Trace Output Base Register (R/W)**
		6:0		Reserved.
		MAXPHYADDR[1]-1:7		**Base physical address.**
		63:MAXPHYADDR		Reserved.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	THREAD	**Trace Output Mask Pointers Register (R/W)**
		6:0		Reserved.
		31:7		**MaskOrTableOffset**
		63:32		**Output Offset.**
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		Reserved, MBZ.
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		Reserved, MBZ
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**Reserved; writing 0 will #GP if also setting TraceEn**
		63:14		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**

Table 2-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		Reserved, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		63:6		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	THREAD	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match

NOTES:

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-33 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 2-33. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**

Table 2-33. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Enable Package C-State Auto-demotion (R/W)**
		30		**Enable Package C-State Undemotion (R/W)**
		63:31		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6core active.
		63:48		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, “PP0/PP1 RAPL Domains.”
See Table 2-18, Table 2-19, Table 2-20, Table 2-23, Table 2-27, Table 2-28, Table 2-32 for other MSR definitions applicable to processors with CPUID signature 06_3DH.				

2.14 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 2-34 are available and common to Intel® Xeon® Processor D product Family (CPUID DisplayFamily_DisplayModel = 06_56H) and to Intel Xeon processors E5 v4, E7 v4 families (CPUID DisplayFamily_DisplayModel = 06_4FH). They are based on the Broadwell microarchitecture.

See Section 2.14.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-24.
		1		Enable_PPIN (R/W) See Table 2-24.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** See Table 2-24.
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** See Table 2-24.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** See Table 2-24.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** See Table 2-24.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** See Table 2-24.
		30	Package	**Programmable TJ OFFSET (R/O)** See Table 2-24.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** See Table 2-24.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		16		**Automatic C-State Conversion Enable (R/W)** If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6)
		24:17		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		Count
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		**Temperature Target (RO)** See Table 2-24.
		27:24		**TCC Activation Offset (R/W)** See Table 2-24.
		63:28		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C**
		15:8	Package	**Maximum Ratio Limit for 2C**
		23:16	Package	**Maximum Ratio Limit for 3C**
		31:24	Package	**Maximum Ratio Limit for 4C**
		39:32	Package	**Maximum Ratio Limit for 5C**
		47:40	Package	**Maximum Ratio Limit for 6C**
		55:48	Package	**Maximum Ratio Limit for 7C**
		63:56	Package	**Maximum Ratio Limit for 8C**
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C**
		15:8	Package	**Maximum Ratio Limit for 10C**

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	**Maximum Ratio Limit for 11C**
		31:24	Package	**Maximum Ratio Limit for 12C**
		39:32	Package	**Maximum Ratio Limit for 13C**
		47:40	Package	**Maximum Ratio Limit for 14C**
		55:48	Package	**Maximum Ratio Limit for 15C**
		63:56	Package	**Maximum Ratio Limit for 16C**
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** Energy consumed by DRAM devices
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**Power Budget Management Status (R0)** When set, frequency is reduced below the operating system request due to PBM limit
		3		**Platform Configuration Services Status (R0)** When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		**Multi-Core Turbo Log** When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		**Core Frequency P1 Log** When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Core Max n-core Turbo Frequency Limiting Log** When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		**Core Frequency Limiting Log** When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP"
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP"
		7:0		**Minimum Performance (R/W)**
		15:8		**Maximum Performance (R/W)**
		23:16		**Desired Performance (R/W)**
		63:24		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback"
		1:0		Reserved.
		2		**Excursion to Minimum (RO)**
		63:3		Reserved.
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x00: no monitoring** **0x01: L3 occupancy monitoring** **0x02: Total memory bandwidth monitoring** **0x03: Local memory bandwidth monitoring** **All other encoding reserved**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W)**
		9:0		**RMID**
		31:10		**Reserved**
		51:32		**COS (R/W).**
		63: 52		**Reserved**
C90H	3216	IA32_L3_QOS_MASK_0	Package	**L3 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		**CBM: Bit vector of available L3 ways for COS 0 enforcement**
		63:20		**Reserved**
C91H	3217	IA32_L3_QOS_MASK_1	Package	**L3 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		**CBM: Bit vector of available L3 ways for COS 1 enforcement**
		63:20		**Reserved**

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C92H	3218	IA32_L3_QOS_MASK_2	Package	**L3 Class Of Service Mask - COS 2 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:19		**CBM: Bit vector of available L3 ways for COS 2 enforcement**
		63:20		**Reserved**
C93H	3219	IA32_L3_QOS_MASK_3	Package	**L3 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L3 ways for COS 3 enforcement**
		63:20		**Reserved**
C94H	3220	IA32_L3_QOS_MASK_4	Package	**L3 Class Of Service Mask - COS 4 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4
		0:19		**CBM: Bit vector of available L3 ways for COS 4 enforcement**
		63:20		**Reserved**
C95H	3221	IA32_L3_QOS_MASK_5	Package	**L3 Class Of Service Mask - COS 5 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		**CBM: Bit vector of available L3 ways for COS 5 enforcement**
		63:20		**Reserved**
C96H	3222	IA32_L3_QOS_MASK_6	Package	**L3 Class Of Service Mask - COS 6 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		**CBM: Bit vector of available L3 ways for COS 6 enforcement**
		63:20		**Reserved**
C97H	3223	IA32_L3_QOS_MASK_7	Package	**L3 Class Of Service Mask - COS 7 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		**CBM: Bit vector of available L3 ways for COS 7 enforcement**
		63:20		**Reserved**
C98H	3224	IA32_L3_QOS_MASK_8	Package	**L3 Class Of Service Mask - COS 8 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8
		0:19		**CBM: Bit vector of available L3 ways for COS 8 enforcement**
		63:20		**Reserved**
C99H	3225	IA32_L3_QOS_MASK_9	Package	**L3 Class Of Service Mask - COS 9 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		**CBM: Bit vector of available L3 ways for COS 9 enforcement**
		63:20		**Reserved**
C9AH	3226	IA32_L3_QOS_MASK_10	Package	**L3 Class Of Service Mask - COS 10 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		**CBM: Bit vector of available L3 ways for COS 10 enforcement**
		63:20		**Reserved**

Table 2-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C9BH	3227	IA32_L3_QOS_MASK_11	Package	**L3 Class Of Service Mask - COS 11 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11
		0:19		**CBM: Bit vector of available L3 ways for COS 11 enforcement**
		63:20		**Reserved**
C9CH	3228	IA32_L3_QOS_MASK_12	Package	**L3 Class Of Service Mask - COS 12 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12
		0:19		**CBM: Bit vector of available L3 ways for COS 12 enforcement**
		63:20		**Reserved**
C9DH	3229	IA32_L3_QOS_MASK_13	Package	**L3 Class Of Service Mask - COS 13 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13
		0:19		**CBM: Bit vector of available L3 ways for COS 13 enforcement**
		63:20		**Reserved**
C9EH	3230	IA32_L3_QOS_MASK_14	Package	**L3 Class Of Service Mask - COS 14 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		**CBM: Bit vector of available L3 ways for COS 14 enforcement**
		63:20		**Reserved**
C9FH	3231	IA32_L3_QOS_MASK_15	Package	**L3 Class Of Service Mask - COS 15 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		**CBM: Bit vector of available L3 ways for COS 15 enforcement**
		63:20		**Reserved**

2.14.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-35 are available to Intel® Xeon® Processor D Product Family (CPUID DisplayFamily_DisplayModel = 06_56H). The Intel® Xeon® processor D product family is based on the Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-18, Table 2-27, Table 2-32, Table 2-34, and Table 2-35.

Table 2-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved

Table 2-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” through Section 15.3.2.4, “IA32_MCi_MISC MSRs.”. Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	

Table 2-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
See Table 2-18, Table 2-27, Table 2-32, and Table 2-34 for other MSR definitions applicable to processors with CPUID signature 06_56H.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.14.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-35 are available to Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID DisplayFamily_DisplayModel = 06_4FH). The Intel® Xeon® processor E5 v4 family is based on the Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-27, Table 2-32, Table 2-34, and Table 2-36.

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
C81H	3201	IA32_L3_QOS_CFG	Package	Cache Allocation Technology Configuration (R/W)
		0		CAT Enable. Set 1 to enable Cache Allocation Technology
		63:1		Reserved.
See Table 2-18, Table 2-19, Table 2-27, and Table 2-28 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.15 MSRS IN THE 6TH GENERATION INTEL® CORE™ PROCESSORS AND 7TH GENERATION INTEL® CORE™ PROCESSORS

6th generation Intel® Core™ processors are based on the Skylake microarchitecture and have CPUID DisplayFamily_DisplayModel signatures of 06_4EH and 06_5EH. 7th Generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and have CPUID DisplayFamily_DisplayModel signatures of 06_8EH and 06_9EH. These processors support the MSR interfaces listed in Table 2-18, Table 2-19, Table 2-23, Table 2-27, Table 2-33, Table 2-37, and Table 2-38. For an MSR listed in Table 2-37 that also appears in the model-specific tables of prior generations, Table 2-37 supercede prior generation tables.

The notation of "Platform" in the Scope column (with respect to MSR_PLATFORM_ENERGY_COUNTER and MSR_PLATFORM_POWER_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor's implementation.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		18		**SGX global functions enable (R/WL)**
		20		**LMCE_ON (R/WL)**
		63:21		Reserved.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	**Dec**			
FEH	254	IA32_MTRRCAP	Thread	**MTRR Capality (RO, Architectural).** See Table 2-2
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		18:2		Reserved.
		19		**Disable Race to Halt Optimization (R/W)** Setting this bit disables the Race to Halt optimization and avoid this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization. Default value is 1 for processors that do not support Race to Halt optimization.
		20		**Disable Energy Efficiency Optimization (R/W)** Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:21		Reserved.
300H	768	MSR_SGXOWNEREPOCH0	Package	**Lower 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	768	MSR_SGXOWNEREPOCH1	Package	**Upper 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.
38EH	910	IA32_PERF_GLOBAL_ STATUS		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Ovf_PMC0**
		1	Thread	**Ovf_PMC1**
		2	Thread	**Ovf_PMC2**
		3	Thread	**Ovf_PMC3**
		4	Thread	**Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		54:35		Reserved.
		55	Thread	**Trace_ToPA_PMI.**
		57:56		Reserved.
		58	Thread	**LBR_Frz.**
		59	Thread	**CTR_Frz.**
		60	Thread	**ASCI.**
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
390H	912	IA32_PERF_GLOBAL_STAT US_RESET		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Set 1 to clear Ovf_PMC0**
		1	Thread	**Set 1 to clear Ovf_PMC1**

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2	Thread	**Set 1 to clear Ovf_PMC2**
		3	Thread	**Set 1 to clear Ovf_PMC3**
		4	Thread	**Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to clear Ovf_FixedCtr0**
		33	Thread	**Set 1 to clear Ovf_FixedCtr1**
		34	Thread	**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55	Thread	**Set 1 to clear Trace_ToPA_PMI.**
		57:56		Reserved.
		58	Thread	**Set 1 to clear LBR_Frz.**
		59	Thread	**Set 1 to clear CTR_Frz.**
		60	Thread	**Set 1 to clear ASCI.**
		61	Thread	**Set 1 to clear Ovf_Uncore**
		62	Thread	**Set 1 to clear Ovf_BufDSSAVE**
		63	Thread	**Set 1 to clear CondChgd**
391H	913	IA32_PERF_GLOBAL_STATUS_SET		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Set 1 to cause Ovf_PMC0 = 1**
		1	Thread	**Set 1 to cause Ovf_PMC1 = 1**
		2	Thread	**Set 1 to cause Ovf_PMC2 = 1**
		3	Thread	**Set 1 to cause Ovf_PMC3 = 1**
		4	Thread	**Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to cause Ovf_FixedCtr0 = 1**
		33	Thread	**Set 1 to cause Ovf_FixedCtr1 = 1**
		34	Thread	**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55	Thread	**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		58	Thread	**Set 1 to cause LBR_Frz = 1**
		59	Thread	**Set 1 to cause CTR_Frz = 1**
		60	Thread	**Set 1 to cause ASCI = 1**
		61	Thread	**Set 1 to cause Ovf_Uncore**
		62	Thread	**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	913	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F7H	1015	MSR_PEBS_FRONTEND	Thread	**FrontEnd Precise Event Condition Select (R/W)**
		2:0		**Event Code Select**
		3		Reserved.
		4		**Event Code Select High**
		7:5		Reserved.
		19:8		**IDQ_Bubble_Length Specifier**
		22:20		**IDQ_Bubble_Width Specifier**
		63:23		**Reserved**
500H	1280	IA32_SGX_SVN_STATUS	Thread	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		15:1		Reserved.
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Thread	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Thread	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ
		48:32		**PacketByteCnt**
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Thread	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Thread	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Thread	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Thread	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Thread	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, “PP0/PP1 RAPL Domains.”
64DH	1613	MSR_PLATFORM_ENERGY_COUNTER	Platform*	Platform Energy Counter. (R/O). This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:0		Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, Included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Enery_Status_Unit.
		63:32		Reserved.
64EH	1614	MSR_PPERF	Thread	Productive Performance Count. (R/O).
		63:0		Hardware's view of workload scalability. See Section 14.4.5.1
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Residency State Regulation Status (R0)** When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).
		7		**VR Therm Design Current Status (R0)** When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		**Other Status (R0)** When set, frequency is reduced below the operating system request due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Residency State Regulation Log** When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
652H	1618	MSR_PKG_HDC_CONFIG	Package	**HDC Configuration (R/W).**
		2:0		**PKG_Cx_Monitor.** **Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY**
		63: 3		**Reserved**
653H	1619	MSR_CORE_HDC_ RESIDENCY	Core	Core HDC Idle Residency. (R/O).
		63:0		Core_Cx_Duty_Cycle_Cnt.
655H	1621	MSR_PKG_HDC_SHALLOW_ RESIDENCY	Package	Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle. (R/O).
		63:0		Pkg_C2_Duty_Cycle_Cnt.
656H	1622	MSR_PKG_HDC_DEEP_ RESIDENCY	Package	Package Cx HDC Idle Residency. (R/O).
		63:0		Pkg_Cx_Duty_Cycle_Cnt.
658H	1624	MSR_WEIGHTED_CORE_C0	Package	Core-count Weighted C0 Residency. (R/O).
		63:0		Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.
659H	1625	MSR_ANY_CORE_C0	Package	Any Core C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.
65AH	1626	MSR_ANY_GFXE_C0	Package	Any Graphics Engine C0 Residency. (R/O)

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.
65BH	1627	MSR_CORE_GFXE_OVERLAP_C0	Package	Core and Graphics Engine Overlapped C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.
65CH	1628	MSR_PLATFORM_POWER_LIMIT	Platform*	**Platform Power Limit Control (R/W-L)** Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.
		14:0		Platform Power Limit #1. Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.
		15		Enable Platform Power Limit #1. When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.
		16		Platform Clamping Limitation #1. When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value. This bit is writeable only when CPUID (EAX=6):EAX[4] is set.
		23:17		Time Window for Platform Power Limit #1. Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17]. The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit].
		31:24		Reserved

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		46:32		Platform Power Limit #2. Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e. Platform Power Limit # 1)
		47		Enable Platform Power Limit #2. When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.
		48		Platform Clamping Limitation #2. When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.
		62:49		Reserved
		63		Lock. Setting this bit will lock all other bits of this MSR until system RESET.
690H	1680	MSR_ LASTBRANCH_16_FROM_IP	Thread	**Last Branch Record 16 From IP (R/W)** One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.11
691H	1681	MSR_ LASTBRANCH_17_FROM_IP	Thread	**Last Branch Record 17 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_ LASTBRANCH_18_FROM_IP	Thread	**Last Branch Record 18 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_ LASTBRANCH_19_FROM_IP	Thread	**Last Branch Record 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Thread	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Thread	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Thread	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Thread	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Thread	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Thread	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Thread	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Thread	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Thread	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Thread	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Thread	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Thread	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W)** **(frequency refers to processor graphics frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		Running Average Thermal Limit Status (R0) When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		12		**Inefficient Operation Status (R0)** When set, processor graphics frequency is operating below target frequency.
		15:13		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Inefficient Operation Log** When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	**Dec**			
		63:29		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W)** **(frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		Reserved.
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:28		Reserved.
6D0H	1744	MSR_ LASTBRANCH_16_TO_IP	Thread	**Last Branch Record 16 To IP (R/W)** One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **destination instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.11
6D1H	1745	MSR_ LASTBRANCH_17_TO_IP	Thread	**Last Branch Record 17 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_ LASTBRANCH_18_TO_IP	Thread	**Last Branch Record 18 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_ LASTBRANCH_19_TO_IP	Thread	**Last Branch Record 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Thread	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Thread	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Thread	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Thread	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Thread	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Thread	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Thread	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Thread	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Thread	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Thread	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Thread	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Thread	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, “Enabling HWP”
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, “HWP Performance Range and Dynamic Capabilities”
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Section 14.4.4, “Managing HWP”
773H	1907	IA32_HWP_INTERRUPT	Thread	See Section 14.4.6, “HWP Notifications”
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, “Managing HWP”
		7:0		**Minimum Performance (R/W).**
		15:8		**Maximum Performance (R/W).**
		23:16		**Desired Performance (R/W).**
		31:24		**Energy/Performance Preference (R/W).**
		41:32		**Activity Window (R/W).**
		42		**Package Control (R/W).**
		63:43		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, “HWP Feedback”
D90H	3472	IA32_BNDCFGS	Thread	See Table 2-2.
DA0H	3488	IA32_XSS	Thread	See Table 2-2.
DB0H	3504	IA32_PKG_HDC_CTL	Package	See Section 14.5.2, “Package level Enabling HDC”
DB1H	3505	IA32_PM_CTL1	Thread	See Section 14.5.3, “Logical-Processor Level HDC Control”
DB2H	3506	IA32_THREAD_STALL	Thread	See Section 14.5.4.1, “IA32_THREAD_STALL”

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC0H	3520	MSR_LBR_INFO_0	Thread	**Last Branch Record 0 Additional Information (R/W)** One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.8.1, "LBR Stack."
DC1H	3521	MSR_LBR_INFO_1	Thread	**Last Branch Record 1 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC2H	3522	MSR_LBR_INFO_2	Thread	**Last Branch Record 2 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC3H	3523	MSR_LBR_INFO_3	Thread	**Last Branch Record 3 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC4H	3524	MSR_LBR_INFO_4	Thread	**Last Branch Record 4 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC5H	3525	MSR_LBR_INFO_5	Thread	**Last Branch Record 5 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC6H	3526	MSR_LBR_INFO_6	Thread	**Last Branch Record 6 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC7H	3527	MSR_LBR_INFO_7	Thread	**Last Branch Record 7 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC8H	3528	MSR_LBR_INFO_8	Thread	**Last Branch Record 8 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC9H	3529	MSR_LBR_INFO_9	Thread	**Last Branch Record 9 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCAH	3530	MSR_LBR_INFO_10	Thread	**Last Branch Record 10 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCBH	3531	MSR_LBR_INFO_11	Thread	**Last Branch Record 11 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCCH	3532	MSR_LBR_INFO_12	Thread	**Last Branch Record 12 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCDH	3533	MSR_LBR_INFO_13	Thread	**Last Branch Record 13 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCEH	3534	MSR_LBR_INFO_14	Thread	**Last Branch Record 14 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCFH	3535	MSR_LBR_INFO_15	Thread	**Last Branch Record 15 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD0H	3536	MSR_LBR_INFO_16	Thread	**Last Branch Record 16 Additional Information (R/W)** See description of MSR_LBR_INFO_0.

Table 2-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture and 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD1H	3537	MSR_LBR_INFO_17	Thread	**Last Branch Record 17 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD2H	3538	MSR_LBR_INFO_18	Thread	**Last Branch Record 18 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD3H	3539	MSR_LBR_INFO_19	Thread	**Last Branch Record 19 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD4H	3520	MSR_LBR_INFO_20	Thread	**Last Branch Record 20 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD5H	3521	MSR_LBR_INFO_21	Thread	**Last Branch Record 21 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD6H	3522	MSR_LBR_INFO_22	Thread	**Last Branch Record 22 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD7H	3523	MSR_LBR_INFO_23	Thread	**Last Branch Record 23 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD8H	3524	MSR_LBR_INFO_24	Thread	**Last Branch Record 24 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD9H	3525	MSR_LBR_INFO_25	Thread	**Last Branch Record 25 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDAH	3526	MSR_LBR_INFO_26	Thread	**Last Branch Record 26 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDBH	3527	MSR_LBR_INFO_27	Thread	**Last Branch Record 27 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDCH	3528	MSR_LBR_INFO_28	Thread	**Last Branch Record 28 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDDH	3529	MSR_LBR_INFO_29	Thread	**Last Branch Record 29 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDEH	3530	MSR_LBR_INFO_30	Thread	**Last Branch Record 30 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDFH	3531	MSR_LBR_INFO_31	Thread	**Last Branch Record 31 Additional Information (R/W)** See description of MSR_LBR_INFO_0.

Table 2-38 lists the MSRs of uncore PMU for Intel processors with CPUID DisplayFamily_DisplayModel signatures of 06_4EH, 06_5EH, 06_8EH and 06_9EH.

Table 2-38. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors and 7th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		43:0		Current count
		63:44		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics),
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
710H	1808	MSR_UNC_CBO_1_PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR

Table 2-38. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors and 7th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
E01H	3585	MSR_UNC_PERF_GLOBAL_ CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
E02H	3586	MSR_UNC_PERF_GLOBAL_ STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.

2.16 MSRS IN FUTURE INTEL® XEON® PROCESSORS

Future Intel® Xeon® Processors (CPUID DisplayFamily_DisplayModel = 06_55H) support the machine check bank registers listed in Table 2-39.

Table 2-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.

Table 2-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC0 reports MC error from the IFU module.
401H	1025	IA32_MC0_STATUS	Core	
402H	1026	IA32_MC0_ADDR	Core	
403H	1027	IA32_MC0_MISC	Core	
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC1 reports MC error from the DCU module.
405H	1029	IA32_MC1_STATUS	Core	
406H	1030	IA32_MC1_ADDR	Core	
407H	1031	IA32_MC1_MISC	Core	
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC2 reports MC error from the DTLB module.
409H	1033	IA32_MC2_STATUS	Core	
40AH	1034	IA32_MC2_ADDR	Core	
40BH	1035	IA32_MC2_MISC	Core	
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC3 reports MC error from the MLC module.
40DH	1037	IA32_MC3_STATUS	Core	
40EH	1038	IA32_MC3_ADDR	Core	
40FH	1039	IA32_MC3_MISC	Core	

Table 2-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_55H

Register Address Hex	Register Address Dec	Register Name	Scope	Bit Description
410H	1040	IA32_MC4_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC4 reports MC error from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from a link interconnect module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the M2M 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the M2M 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC12 report MC error from each channel of a link interconnect module.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	

Table 2-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	

2.17 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily_DisplayModel signature 06_57H, supports the MSR interfaces listed in Table 2-40. These processors are based on the Knights Landing microarchitecture. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_ SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_ COUNTER	Thread	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)**
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Reserved**
		2		**Enable VMX outside SMX operation (R/WL)**
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		**Protected Processor Inventory Number (R/O)** A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	THREAD	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	THREAD	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	THREAD	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Package	**C-State Configuration Control (R/W)**

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest C-state for the package. This feature does not limit the processor core C-state. The power-on default value from bit[2:0] of this register reports the deepest package C-state the processor is capable to support when manufactured. It is recommended that BIOS always read the power-on default value reported from this bit field to determine the supported deepest C-state on the processor and leave it as default without changing it. 000b - C0/C1 (No package C-state support) 001b - C2 010b - C6 (non retention)* 011b - C6 (Retention)* 100b - Reserved 101b - Reserved 110b - Reserved 111b - No package C-state limit. All C-States supported by the processor are available. Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO registers at MSR_PMG_IO_CAPTURE_BASE[15:0] to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (RO)** When set, locks bits [15:0] of this register for further writes until the next reset occurs.
		25		Reserved.
		26		**C1 State Auto Demotion Enable (R/W)** When set, processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Reserved.
		28		**C1 State Auto Undemotion Enable (R/W)** When set, enables Undemotion from Demoted C1.
		29		**PKG C-State Auto Demotion Enable (R/W)** When set, enables Package C state demotion.
		63:30		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Tile	**Power Management IO Capture Base (R/W)**

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address Hex	Address Dec	Register Name	Scope	Bit Description
		15:0		**LVL_2 Base Address (R/W)** Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.
		22:16		**C-State Range (R/W)** The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.
		63:23		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	**Memory Type Range Register (R)** See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
140H	320	MISC_FEATURE_ENABLES	Thread	**MISC_FEATURE_ENABLES**
		0		Reserved.
		1		**User Mode MONITOR and MWAIT (R/W)** If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL > 0 or in virtual-8086 mode. If MWAIT is executed when CPL > 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	See Table 2-2.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	Thread	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		31:0		**Bank Support (SMM-RO)** One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA).
		55:32		Reserved.
		56		**Targeted SMI (SMM-RO)** Set if targeted SMI is supported.
		57		**SMM_CPU_SVRSTR (SMM-RO)** Set if SMM SRAM save/restore feature is supported.
		58		**SMM_CODE_ACCESS_CHK (SMM-RO)** Set if SMM code access check feature is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved.
186H	390	IA32_PERFEVTSEL0	Thread	Performance Monitoring Event Select Register (R/W) See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_INTERRUPT	Module	**Thermal Interrupt Control (R/W)** See Table 2-2.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
19CH	412	IA32_THERM_STATUS	Module	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)**
		1		**Thermal status log (R/WC0)**
		2		**PROTCHOT # or FORCEPR# status (RO)**
		3		**PROTCHOT # or FORCEPR# log (R/WC0)**
		4		**Critical Temperature status (RO)**
		5		**Critical Temperature status log (R/WC0)**
		6		**Thermal threshold #1 status (RO)**
		7		**Thermal threshold #1 log (R/WC0)**
		8		**Thermal threshold #2 status (RO)**
		9		**Thermal threshold #2 log (R/WC0)**
		10		**Power Limitation status (RO)**
		11		**Power Limitation log (R/WC0)**
		15:12		Reserved.
		22:16		**Digital Readout (RO)**
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)**
		31		**Reading Valid (RO)**
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE	Thread	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable**
		2:1		Reserved.
		3		**Automatic Thermal Control Circuit Enable (R/W)**
		6:4		Reserved.
		7		**Performance Monitoring Available (R)**
		10:8		Reserved.
		11		**Branch Trace Storage Unavailable (RO)**
		12		**Processor Event Based Sampling Unavailable (RO)**
		15:13		Reserved.
		16		**Enhanced Intel SpeedStep Technology Enable (R/W)**
		18		**ENABLE MONITOR FSM (R/W)**
		21:19		Reserved.
		22		**Limit CPUID Maxval (R/W)**
		23		**xTPR Message Disable (R/W)**
		33:24		Reserved.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**XD Bit Disable (R/W)**
		37:35		Reserved.
		38		**Turbo Mode Disable (R/W)**
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		**Temperature Target (R)**
		29:24		**Target Offset (R/W)**
		63:30		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher.
		1	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher.
		63:2		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Shared	**Offcore Response Event Select Register (R/W)**
1A7H	423	MSR_OFFCORE_RSP_1	Shared	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode for Groups of Cores (RW)**
		0		**Reserved**
		7:1	Package	**Maximum Number of Cores in Group 0** Number active processor cores which operates under the maximum ratio limit for group 0.
		15:8	Package	**Maximum Ratio Limit for Group 0** Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.
		20:16	Package	**Number of Incremental Cores Added to Group 1** Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = "group 0 Max ratio limit" - "group ratio delta for group 1".
		23:21	Package	**Group Ratio Delta for Group 1** An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.
		28:24	Package	**Number of Incremental Cores Added to Group 2** Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = "group 1 Max ratio limit" - "group ratio delta for group 2".

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:29	Package	**Group Ratio Delta for Group 2** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.
		36:32	Package	**Number of Incremental Cores Added to Group 3** Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = "group 2 Max ratio limit" - "group ratio delta for group 3".
		39:37	Package	**Group Ratio Delta for Group 3** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.
		44:40	Package	**Number of Incremental Cores Added to Group 4** Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = "group 3 Max ratio limit" - "group ratio delta for group 4".
		47:45	Package	**Group Ratio Delta for Group 4** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.
		52:48	Package	**Number of Incremental Cores Added to Group 5** Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = "group 4 Max ratio limit" - "group ratio delta for group 5".
		55:53	Package	**Group Ratio Delta for Group 5** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.
		60:56	Package	**Number of Incremental Cores Added to Group 6** Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = "group 5 Max ratio limit" - "group ratio delta for group 6".
		63:61	Package	**Group Ratio Delta for Group 6** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.
1B0H	432	IA32_ENERGY_PERF_BIAS	Thread	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)** See Section 17.8.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)**
		0		**LBR** Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.
		1		**BTF** Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.
		5:2		Reserved.
		6		**TR** Setting this bit to 1 enables branch trace messages to be sent.
		7		**BTS** Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.
		8		**BTINT** When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.
		9		**BTS_OFF_OS** When set, BTS or BTM is skipped if CPL = 0.
		10		**BTS_OFF_USR** When set, BTS or BTM is skipped if CPL > 0.
		11		**FREEZE_LBRS_ON_PMI** When set, the LBR stack is frozen on a PMI request.
		12		**FREEZE_PERFMON_ON_PMI** When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.
		13		Reserved.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		14		**FREEZE_WHILE_SMM_EN** When set, freezes perfmon and trace messages while in SMM.
		31:15		Reserved.
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)**
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)**
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Core	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Package	See Table 2-2. See Section 17.4.1, “IA32_DEBUGCTL MSR.”
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Table 2-2.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O)
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	
		63:0		Package C6 Residency Counter. (R/O)
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	
		63:0		Package C7 Residency Counter. (R/O)
3FCH	1020	MSR_MC0_RESIDENCY	Module	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Module C0 Residency Counter. (R/O)
3FDH	1021	MSR_MC6_RESIDENCY	Module	
		63:0		Module C6 Residency Counter. (R/O)
3FFH	1023	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O)
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, “RAPL Interfaces.”
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, “RAPL Interfaces.”
		63:20		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter. (R/O)
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, “Package RAPL Domain.”
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, “Package RAPL Domain.”
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, “Package RAPL Domain.”

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	**Base TDP Ratio (R/O)** See Table 2-23
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O). See Table 2-23
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O). See Table 2-23
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	**ConfigTDP Control (R/W)** See Table 2-23
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)** See Table 2-23
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)**
		1		**Thermal Status (R0)**
		5:2		Reserved.
		6		**VR Therm Alert Status (R0)**
		7		Reserved.
		8		**Electrical Design Point Status (R0)**
		63:9		Reserved.
6E0H	1760	IA32_TSC_DEADLINE	Core	**TSC Target of Local APIC's TSC Deadline Mode (R/W)** See Table 2-2
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)

Table 2-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_57H

Address		Register Name	Scope	Bit Description
Hex	Dec			
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_ 0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W)** See Table 2-2

2.18 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-41 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 2-1.

- MSRs with an "IA32_" prefix are designated as "architectural." This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an "MSR_" prefix are model specific with respect to address functionalities. The column "Model Availability" lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique [1]	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_LINE_ SIZE	3, 4, 6	Shared	See Section 8.10.5, "Monitor/Mwait Address Range Determination."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique [1]	Bit Description
Hex	Dec				
10H	16	IA32_TIME_STAMP_COUNTER	0, 1, 2, 3, 4, 6	Unique	**Time Stamp Counter** See Table 2-2.
					On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.
17H	23	IA32_PLATFORM_ID	0, 1, 2, 3, 4, 6	Shared	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	0, 1, 2, 3, 4, 6	Unique	**APIC Location and Status (R/W)** See Table 2-2. See Section 10.4.4, "Local APIC Status and Location."
2AH	42	MSR_EBC_HARD_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0			**Output Tri-state Enabled (R)** Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		1			**Execute BIST (R)** Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		2			**In Order Queue Depth (R)** Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		3			**MCERR# Observation Disabled (R)** Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
		4			**BINIT# Observation Enabled (R)** Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		6:5			**APIC Cluster ID (R)** Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		7			**Bus Park Disable (R)** Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		11:8			Reserved.
		13:12			**Agent ID (R)** Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		63:14			Reserved.
2BH	43	MSR_EBC_SOFT_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Soft Power-On Configuration (R/W)** Enables and disables processor features.
		0			**RCNT/SCNT On Request Encoding Enable (R/W)** Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).
		1			**Data Error Checking Disable (R/W)** Set to disable system data bus parity checking; clear to enable parity checking.
		2			**Response Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		3			**Address/Request Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		4			**Initiator MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator bus requests (default); clear to enable.
		5			**Internal MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator internal errors (default); clear to enable.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
		6			**BINIT# Driver Disable (R/W)** Set to disable BINIT# driver (default); clear to enable driver.
		63:7			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	2,3, 4, 6	Shared	**Processor Frequency Configuration** The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.
		15:0			Reserved.
		18:16			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)
					133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
					266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.
		23:19			Reserved.
		31:24			**Core Clock Frequency to System Bus Frequency Ratio (R)** The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			Reserved.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
2CH	44	MSR_EBC_FREQUENCY_ID	0, 1	Shared	**Processor Frequency Configuration (R)** The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.
		20:0			Reserved.
		23:21			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz All others values reserved.
		63:24			Reserved.
3AH	58	IA32_FEATURE_CONTROL	3, 4, 6	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2 (If CPUID.01H:ECX.[bit 5])
79H	121	IA32_BIOS_UPDT_TRIG	0, 1, 2, 3, 4, 6	Shared	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	0, 1, 2, 3, 4, 6	Unique	**BIOS Update Signature ID (R/W)** See Table 2-2.
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	**SMM Monitor Configuration (R/W)** See Table 2-2.
FEH	254	IA32_MTRRCAP	0, 1, 2, 3, 4, 6	Unique	**MTRR Information** See Section 11.11.1, "MTRR Feature Identification.".
174H	372	IA32_SYSENTER_CS	0, 1, 2, 3, 4, 6	Unique	**CS register target for CPL 0 code (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
175H	373	IA32_SYSENTER_ESP	0, 1, 2, 3, 4, 6	Unique	**Stack pointer for CPL 0 stack (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
176H	374	IA32_SYSENTER_EIP	0, 1, 2, 3, 4, 6	Unique	**CPL 0 code entry point (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
179H	377	IA32_MCG_CAP	0, 1, 2, 3, 4, 6	Unique	**Machine Check Capabilities (R)** See Table 2-2. See Section 15.3.1.1, "IA32_MCG_CAP MSR."
17AH	378	IA32_MCG_STATUS	0, 1, 2, 3, 4, 6	Unique	**Machine Check Status. (R)** See Table 2-2. See Section 15.3.1.2, "IA32_MCG_STATUS MSR."
17BH	379	IA32_MCG_CTL			**Machine Check Feature Enable (R/W)** See Table 2-2. See Section 15.3.1.3, "IA32_MCG_CTL MSR."
180H	384	MSR_MCG_RAX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EAX/RAX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
181H	385	MSR_MCG_RBX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EBX/RBX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
182H	386	MSR_MCG_RCX	0, 1, 2, 3, 4, 6	Unique	**Machine Check ECX/RCX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
183H	387	MSR_MCG_RDX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDX/RDX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
184H	388	MSR_MCG_RSI	0, 1, 2, 3, 4, 6	Unique	**Machine Check ESI/RSI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
185H	389	MSR_MCG_RDI	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDI/RDI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
186H	390	MSR_MCG_RBP	0, 1, 2, 3, 4, 6	Unique	**Machine Check EBP/RBP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
187H	391	MSR_MCG_RSP	0, 1, 2, 3, 4, 6	Unique	**Machine Check ESP/RSP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
188H	392	MSR_MCG_RFLAGS	0, 1, 2, 3, 4, 6	Unique	**Machine Check EFLAGS/RFLAG Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
189H	393	MSR_MCG_RIP	0, 1, 2, 3, 4, 6	Unique	**Machine Check EIP/RIP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
18AH	394	MSR_MCG_MISC	0, 1, 2, 3, 4, 6	Unique	**Machine Check Miscellaneous** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		0			**DS** When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.
		63:1			Reserved.
18BH - 18FH	395	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved.
190H	400	MSR_MCG_R8	0, 1, 2, 3, 4, 6	Unique	**Machine Check R8** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
191H	401	MSR_MCG_R9	0, 1, 2, 3, 4, 6	Unique	**Machine Check R9D/R9** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
192H	402	MSR_MCG_R10	0, 1, 2, 3, 4, 6	Unique	**Machine Check R10** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
193H	403	MSR_MCG_R11	0, 1, 2, 3, 4, 6	Unique	**Machine Check R11** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
194H	404	MSR_MCG_R12	0, 1, 2, 3, 4, 6	Unique	**Machine Check R12** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
195H	405	MSR_MCG_R13	0, 1, 2, 3, 4, 6	Unique	**Machine Check R13** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
196H	406	MSR_MCG_R14	0, 1, 2, 3, 4, 6	Unique	**Machine Check R14** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
197H	407	MSR_MCG_R15	0, 1, 2, 3, 4, 6	Unique	**Machine Check R15** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
198H	408	IA32_PERF_STATUS	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
199H	409	IA32_PERF_CTL	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
19AH	410	IA32_CLOCK_MODULATION	0, 1, 2, 3, 4, 6	Unique	**Thermal Monitor Control (R/W)** See Table 2-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	**Thermal Interrupt Control (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19CH	412	IA32_THERM_STATUS	0, 1, 2, 3, 4, 6	Shared	**Thermal Monitor Status (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19DH	413	MSR_THERM2_CTL			Thermal Monitor 2 Control.
			3,	Shared	For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.
			4, 6	Shared	For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4, 6	Shared	**Enable Miscellaneous Processor Features (R/W)**
		0			Fast-Strings Enable. See Table 2-2.
		1			Reserved.
		2			**x87 FPU Fopcode Compatibility Mode Enable**

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/Unique[1]	Bit Description
		3			**Thermal Monitor 1 Enable** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
		4			**Split-Lock Disable** When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
					This debug feature is specific to the Pentium 4 processor.
		5			Reserved.
		6			**Third-Level Cache Disable (R/W)** When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache. Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs. See Section 11.5.4, "Disabling and Enabling the L3 Cache."
		7			**Performance Monitoring Available (R)** See Table 2-2.
		8			**Suppress Lock Enable** When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			**Prefetch Queue Disable** When set, disables the prefetch queue. When clear (default), enables the prefetch queue.
		10			**FERR# Interrupt Reporting Enable (R/W)** When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.
					When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Availability	Shared/ Unique[1]	Bit Description
					This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.
		11			**Branch Trace Storage Unavailable (BTS_UNAVILABLE) (R)** See Table 2-2. When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.
		12			**PEBS_UNAVILABLE: Processor Event Based Sampling Unavailable (R)** See Table 2-2. When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.
		13	3		**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		17:14			Reserved.
		18	3, 4, 6		**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19			**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector.
					Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			Reserved.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
		22	3, 4, 6		**Limit CPUID MAXVAL (R/W)** See Table 2-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.
		23		Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		24			**L1 Data Cache Context Mode (R/W)** When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 11.5.6, "L1 Data Cache Context Mode." When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].
		33:25			Reserved.
		34		Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35			Reserved.
1A1H	417	MSR_PLATFORM_BRV	3, 4, 6	Shared	**Platform Feature Requirements (R)**
		17:0			Reserved.
		18			**PLATFORM Requirements** When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
		63:19			Reserved.
1D7H	471	MSR_LER_FROM_LIP	0, 1, 2, 3, 4, 6	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.12.3, "Last Exception Records."
		31:0			**From Linear IP** Linear address of the last branch instruction.
		63:32			Reserved.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique [1]	Bit Description
Hex	Dec				
1D7H	471	63:0		Unique	**From Linear IP** Linear address of the last branch instruction (If IA-32e mode is active).
1D8H	472	MSR_LER_TO_LIP	0, 1, 2, 3, 4, 6	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.12.3, “Last Exception Records.”
		31:0			**From Linear IP** Linear address of the target of the last branch instruction.
		63:32			Reserved.
1D8H	472	63:0		Unique	**From Linear IP** Linear address of the target of the last branch instruction (If IA-32e mode is active).
1D9H	473	MSR_DEBUGCTLA	0, 1, 2, 3, 4, 6	Unique	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.12.1, “MSR_DEBUGCTLA MSR.”
1DAH	474	MSR_LASTBRANCH _TOS	0, 1, 2, 3, 4, 6	Unique	**Last Branch Record Stack TOS (R/O)** Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 17.12.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture”; and addresses 1DBH-1DEH and 680H-68FH.
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	**Last Branch Record 0 (R/O)** One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH.
					See Section 17.11, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture.”
1DCH	477	MSR_LASTBRANCH_1	0, 1, 2	Unique	**Last Branch Record 1** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	**Last Branch Record 2** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	**Last Branch Record 3** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Base MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs”.
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20AH	522	IA32_MTRR_PHYSBASE5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20BH	523	IA32_MTRR_PHYSMASK5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20CH	524	IA32_MTRR_PHYSBASE6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20DH	525	IA32_MTRR_PHYSMASK6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20EH	526	IA32_MTRR_PHYSBASE7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
20FH	527	IA32_MTRR_PHYSMASK7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, “Variable Range MTRRs.”
250H	592	IA32_MTRR_FIX64K_00000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, “Fixed Range MTRRs.”

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
258H	600	IA32_MTRR_FIX16K_80000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
259H	601	IA32_MTRR_FIX16K_A0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
268H	616	IA32_MTRR_FIX4K_C0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
269H	617	IA32_MTRR_FIX4K_C8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26AH	618	IA32_MTRR_FIX4K_D0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26BH	619	IA32_MTRR_FIX4K_D8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26CH	620	IA32_MTRR_FIX4K_E0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26DH	621	IA32_MTRR_FIX4K_E8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26EH	622	IA32_MTRR_FIX4K_F0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26FH	623	IA32_MTRR_FIX4K_F8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
277H	631	IA32_PAT	0, 1, 2, 3, 4, 6	Unique	**Page Attribute Table** See Section 11.11.2.2, "Fixed Range MTRRs."
2FFH	767	IA32_MTRR_DEF_TYPE	0, 1, 2, 3, 4, 6	Shared	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
301H	769	MSR_BPU_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
302H	770	MSR_BPU_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
303H	771	MSR_BPU_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
304H	772	MSR_MS_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
305H	773	MSR_MS_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
306H	774	MSR_MS_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
307H	775	MSR_MS_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
308H	776	MSR_FLAME_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
309H	777	MSR_FLAME_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30AH	778	MSR_FLAME_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30BH	779	MSR_FLAME_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30CH	780	MSR_IQ_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30DH	781	MSR_IQ_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30EH	782	MSR_IQ_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
30FH	783	MSR_IQ_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
310H	784	MSR_IQ_COUNTER4	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
311H	785	MSR_IQ_COUNTER5	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.2, "Performance Counters."
360H	864	MSR_BPU_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
361H	865	MSR_BPU_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
362H	866	MSR_BPU_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
363H	867	MSR_BPU_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
364H	868	MSR_MS_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
365H	869	MSR_MS_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
366H	870	MSR_MS_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
367H	871	MSR_MS_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
368H	872	MSR_FLAME_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
369H	873	MSR_FLAME_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
36AH	874	MSR_FLAME_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
36BH	875	MSR_FLAME_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
36CH	876	MSR_IQ_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
36DH	877	MSR_IQ_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
36EH	878	MSR_IQ_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
36FH	879	MSR_IQ_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
370H	880	MSR_IQ_CCCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
371H	881	MSR_IQ_CCCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.3, "CCCR MSRs."
3A0H	928	MSR_BSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A1H	929	MSR_BSU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A2H	930	MSR_FSB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A3H	931	MSR_FSB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A4H	932	MSR_FIRM_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A5H	933	MSR_FIRM_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A6H	934	MSR_FLAME_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A7H	935	MSR_FLAME_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A8H	936	MSR_DAC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3A9H	937	MSR_DAC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3AAH	938	MSR_MOB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3ABH	939	MSR_MOB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3ACH	940	MSR_PMH_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3ADH	941	MSR_PMH_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3AEH	942	MSR_SAAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
3AFH	943	MSR_SAAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B0H	944	MSR_U2L_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B1H	945	MSR_U2L_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B2H	946	MSR_BPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B3H	947	MSR_BPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B4H	948	MSR_IS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B5H	949	MSR_IS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B6H	950	MSR_ITLB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B7H	951	MSR_ITLB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B8H	952	MSR_CRU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3B9H	953	MSR_CRU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3BAH	954	MSR_IQ_ESCR0	0, 1, 2	Shared	See Section 18.15.1, “ESCR MSRs.” This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.
3BBH	955	MSR_IQ_ESCR1	0, 1, 2	Shared	See Section 18.15.1, “ESCR MSRs.” This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.
3BCH	956	MSR_RAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3BDH	957	MSR_RAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3BEH	958	MSR_SSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3C0H	960	MSR_MS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3C1H	961	MSR_MS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3C2H	962	MSR_TBPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”
3C3H	963	MSR_TBPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, “ESCR MSRs.”

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
3C4H	964	MSR_TC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3C5H	965	MSR_TC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3C8H	968	MSR_IX_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3C9H	969	MSR_IX_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3CAH	970	MSR_ALF_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3CBH	971	MSR_ALF_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3CCH	972	MSR_CRU_ESCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3CDH	973	MSR_CRU_ESCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3E0H	992	MSR_CRU_ESCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3E1H	993	MSR_CRU_ESCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3F0H	1008	MSR_TC_PRECISE_EVENT	0, 1, 2, 3, 4, 6	Shared	See Section 18.15.1, "ESCR MSRs."
3F1H	1009	MSR_PEBS_ENABLE	0, 1, 2, 3, 4, 6	Shared	**Processor Event Based Sampling (PEBS) (R/W)** Controls the enabling of processor event sampling and replay tagging.
		12:0			See Table 19-34.
		23:13			Reserved.
		24			**UOP Tag** Enables replay tagging when set.
		25			**ENABLE_PEBS_MY_THR (R/W)** Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.16.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is called ENABLE_PEBS in IA-32 processors that do not support Intel Hyper-Threading Technology.
		26			**ENABLE_PEBS_OTH_THR (R/W)** Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.16.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is reserved for IA-32 processors that do not support Intel Hyper-Threading Technology.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique [1]	Bit Description
Hex	Dec				
		63:27			Reserved.
3F2H	1010	MSR_PEBS_MATRIX_VERT	0, 1, 2, 3, 4, 6	Shared	See Table 19-34.
400H	1024	IA32_MC0_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC0_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC		Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
40AH	1034	IA32_MC2_ADDR			See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC			See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	IA32_MC3_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40EH	1038	IA32_MC3_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
411H	1041	IA32_MC4_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
412H	1042	IA32_MC4_ADDR			See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
413H	1043	IA32_MC4_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
483H	1155	IA32_VMX_EXIT_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls," and see Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls," and see Table 2-2.
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data," and see Table 2-2.
486H	1158	IA32_VMX_CR0_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
487H	1159	IA32_VMX_CR0_FIXED1	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
488H	1160	IA32_VMX_CR4_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
489H	1161	IA32_VMX_CR4_FIXED1	3, 4, 6	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	3, 4, 6	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration," and see Table 2-2.
48BH	1163	IA32_VMX_PROCBASED_CTLS2	3, 4, 6	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
600H	1536	IA32_DS_AREA	0, 1, 2, 3, 4, 6	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	3, 4, 6	Unique	**Last Branch Record 0 (R/W)** One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the **source instruction** for one of the last 16 branches, exceptions, or interrupts taken by the processor.
					The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH.which performed the same function for early releases. See Section 17.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
681H	1665	MSR_LASTBRANCH_1_FROM_IP	3, 4, 6	Unique	**Last Branch Record 1** See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	3, 4, 6	Unique	**Last Branch Record 2** See description of MSR_LASTBRANCH_0 at 680H.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	3, 4, 6	Unique	**Last Branch Record 3** See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	3, 4, 6	Unique	**Last Branch Record 4** See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	3, 4, 6	Unique	**Last Branch Record 5** See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	3, 4, 6	Unique	**Last Branch Record 6** See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	3, 4, 6	Unique	**Last Branch Record 7** See description of MSR_LASTBRANCH_0 at 680H.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique [1]	Bit Description
Hex	Dec				
688H	1672	MSR_LASTBRANCH_8_FROM_IP	3, 4, 6	Unique	**Last Branch Record 8** See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	3, 4, 6	Unique	**Last Branch Record 9** See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	3, 4, 6	Unique	**Last Branch Record 10** See description of MSR_LASTBRANCH_0 at 680H.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	3, 4, 6	Unique	**Last Branch Record 0 (R/W)** One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 17.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	3, 4, 6	Unique	**Last Branch Record 1** See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	3, 4, 6	Unique	**Last Branch Record 2** See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	3, 4, 6	Unique	**Last Branch Record 3** See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	3, 4, 6	Unique	**Last Branch Record 4** See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	3, 4, 6	Unique	**Last Branch Record 5** See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	3, 4, 6	Unique	**Last Branch Record 6** See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	3, 4, 6	Unique	**Last Branch Record 7** See description of MSR_LASTBRANCH_0 at 6C0H.

Table 2-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	3, 4, 6	Unique	**Last Branch Record 8** See description of MSR_LASTBRANCH_0 at 6C0H.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	**Last Branch Record 9** See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	**Last Branch Record 10** See description of MSR_LASTBRANCH_0 at 6C0H.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 6C0H.
C000_ 0080H		IA32_EFER	3, 4, 6	Unique	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	3, 4, 6	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	3, 4, 6	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	3, 4, 6	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	3, 4, 6	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	3, 4, 6	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	3, 4, 6	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

NOTES

1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.

2.18.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-42 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

Table 2-42. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique	Bit Description
107CCH		MSR_IFSB_BUSQ0	3, 4	Shared	**IFSB BUSQ Event Control and Counter Register (R/W)** See Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache.”
107CDH		MSR_IFSB_BUSQ1	3, 4	Shared	**IFSB BUSQ Event Control and Counter Register (R/W)**
107CEH		MSR_IFSB_SNPQ0	3, 4	Shared	**IFSB SNPQ Event Control and Counter Register (R/W)** See Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache.”
107CFH		MSR_IFSB_SNPQ1	3, 4	Shared	**IFSB SNPQ Event Control and Counter Register (R/W)**
107D0H		MSR_EFSB_DRDY0	3, 4	Shared	**EFSB DRDY Event Control and Counter Register (R/W)** See Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache” for details.
107D1H		MSR_EFSB_DRDY1	3, 4	Shared	**EFSB DRDY Event Control and Counter Register (R/W)**
107D2H		MSR_IFSB_CTL6	3, 4	Shared	**IFSB Latency Event Control Register (R/W)** See Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache” for details.
107D3H		MSR_IFSB_CNTR7	3, 4	Shared	**IFSB Latency Event Counter Register (R/W)** See Section 18.21, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache.”

The MSRs listed in Table 2-43 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 2-43 are shared between logical processors in the same core, but are replicated for each core.

Table 2-43. MSRs Unique to Intel® Xeon® Processor 7100 Series

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique	Bit Description
107CCH		MSR_EMON_L3_CTR_CTL0	6	Shared	**GBUSQ Event Control and Counter Register (R/W)** See Section 18.21, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CDH		MSR_EMON_L3_CTR_CTL1	6	Shared	**GBUSQ Event Control and Counter Register (R/W)**
107CEH		MSR_EMON_L3_CTR_CTL2	6	Shared	**GSNPQ Event Control and Counter Register (R/W)** See Section 18.21, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_EMON_L3_CTR_CTL3	6	Shared	**GSNPQ Event Control and Counter Register (R/W)**
107D0H		MSR_EMON_L3_CTR_CTL4	6	Shared	**FSB Event Control and Counter Register (R/W)** See Section 18.21, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache" for details.
107D1H		MSR_EMON_L3_CTR_CTL5	6	Shared	**FSB Event Control and Counter Register (R/W)**
107D2H		MSR_EMON_L3_CTR_CTL6	6	Shared	**FSB Event Control and Counter Register (R/W)**
107D3H		MSR_EMON_L3_CTR_CTL7	6	Shared	**FSB Event Control and Counter Register (R/W)**

2.19 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-44. The column "Shared/Unique" applies to Intel Core Duo processor. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
1H	1	P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and see Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, “Local APIC Status and Location,” and see Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		6: 5		Reserved
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		11		Reserved
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		13		Reserved
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		**APIC Cluster ID (R/O)**

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
		18		**System Bus Frequency (R/O)** 0 = 100 MHz 1 = Reserved
		19		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)**
		26:22		**Clock Frequency Ratio (R/O)**
3AH	58	IA32_FEATURE_CONTROL	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2.
40H	64	MSR_LASTBRANCH_0	Unique	**Last Branch Record 0 (R/W)** One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the 'to' address. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.14, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."
41H	65	MSR_LASTBRANCH_1	Unique	**Last Branch Record 1 (R/W)** See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	Unique	**Last Branch Record 2 (R/W)** See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	Unique	**Last Branch Record 3 (R/W)** See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	Unique	**Last Branch Record 4 (R/W)** See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	Unique	**Last Branch Record 5 (R/W)** See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	Unique	**Last Branch Record 6 (R/W)** See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	Unique	**Last Branch Record 7 (R/W)** See description of MSR_LASTBRANCH_0.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Unique	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance counter register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed (RO)** This field indicates the scaleable bus clock speed:

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count. (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count. (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	
		0		**RIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		**EIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_ MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_ INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	
		15:0		Reserved.
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Miscellaneous Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		2:0		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address Hex	Register Address Dec	Register Name	Shared/ Unique	Bit Description
		9:8		Reserved.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12		Reserved.
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** 1 = Enhanced Intel SpeedStep Technology enabled
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19		**Reserved.**
		22	Shared	**Limit CPUID Maxval (R/W)** See Table 2-2. Setting this bit may cause behavior in software that depends on the availability of CPUID leaves greater than 2.
		33:23		Reserved.
		34	Shared	**XD Bit Disable (R/W)** See Table 2-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1E0H	480	ROB_CR_ BKUPTMPDR6	Unique	
		1:0		Reserved.
		2		Fast String Enable bit. (Default, enabled)
200H	512	MTRRphysBase0	Unique	
201H	513	MTRRphysMask0	Unique	
202H	514	MTRRphysBase1	Unique	
203H	515	MTRRphysMask1	Unique	
204H	516	MTRRphysBase2	Unique	
205H	517	MTRRphysMask2	Unique	
206H	518	MTRRphysBase3	Unique	
207H	519	MTRRphysMask3	Unique	
208H	520	MTRRphysBase4	Unique	
209H	521	MTRRphysMask4	Unique	
20AH	522	MTRRphysBase5	Unique	
20BH	523	MTRRphysMask5	Unique	
20CH	524	MTRRphysBase6	Unique	
20DH	525	MTRRphysMask6	Unique	
20EH	526	MTRRphysBase7	Unique	
20FH	527	MTRRphysMask7	Unique	
250H	592	MTRRfix64K_00000	Unique	
258H	600	MTRRfix16K_80000	Unique	
259H	601	MTRRfix16K_A0000	Unique	
268H	616	MTRRfix4K_C0000	Unique	
269H	617	MTRRfix4K_C8000	Unique	
26AH	618	MTRRfix4K_D0000	Unique	
26BH	619	MTRRfix4K_D8000	Unique	
26CH	620	MTRRfix4K_E0000	Unique	

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
26DH	621	MTRRfix4K_E8000	Unique	
26EH	622	MTRRfix4K_F0000	Unique	
26FH	623	MTRRfix4K_F8000	Unique	
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MC0_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	MSR_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	MSR_MC3_MISC	Unique	

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
414H	1044	MSR_MC5_CTL	Unique	
415H	1045	MSR_MC5_STATUS	Unique	
416H	1046	MSR_MC5_ADDR	Unique	
417H	1047	MSR_MC5_MISC	Unique	
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, “Basic VMX Information” (If CPUID.01H:ECX.[bit 9])
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3, “VM-Execution Controls” (If CPUID.01H:ECX.[bit 9])
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, “VM-Execution Controls” (If CPUID.01H:ECX.[bit 9])
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, “VM-Exit Controls” (If CPUID.01H:ECX.[bit 9])
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, “VM-Entry Controls” (If CPUID.01H:ECX.[bit 9])
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, “Miscellaneous Data” (If CPUID.01H:ECX.[bit 9])
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, “VMX-Fixed Bits in CR0” (If CPUID.01H:ECX.[bit 9])
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, “VMX-Fixed Bits in CR0” (If CPUID.01H:ECX.[bit 9])
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, “VMX-Fixed Bits in CR4” (If CPUID.01H:ECX.[bit 9])
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, “VMX-Fixed Bits in CR4” (If CPUID.01H:ECX.[bit 9])
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, “VMCS Enumeration” (If CPUID.01H:ECX.[bit 9])

Table 2-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9] and IA32_VMX_PROCBASED_CTLS[bit 63])
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.15.4, "Debug Store (DS) Mechanism."
		31:0		**DS Buffer Management Area** Linear address of the first byte of the DS buffer management area.
		63:32		Reserved.
C000_ 0080H		IA32_EFER	Unique	See Table 2-2.
		10:0		Reserved.
		11		**Execute Disable Bit Enable**
		63:12		Reserved.

2.20 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.21 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

Table 2-45. MSRs in Pentium M Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.16, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
2AH	42	MSR_EBL_CR_POWERON	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features. **(R)** Indicates current processor configuration.
		0	Reserved.
		1	**Data Error Checking Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		2	**Response Error Checking Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		3	**MCERR# Drive Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		4	**Address Parity Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		6:5	Reserved.
		7	**BINIT# Driver Enable (R)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		8	**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9	**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10	**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		11	Reserved.
		12	**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		13	Reserved.
		14	**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes Always 0 on the Pentium M processor.
		15	Reserved.
		17:16	**APIC Cluster ID (R/O)** Always 00B on the Pentium M processor.
		18	**System Bus Frequency (R/O)** 0 = 100 MHz 1 = Reserved Always 0 on the Pentium M processor.
		19	Reserved.
		21: 20	**Symmetric Arbitration ID (R/O)** Always 00B on the Pentium M processor.
		26:22	Clock Frequency Ratio (R/O)

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
40H	64	MSR_LASTBRANCH_0	**Last Branch Record 0 (R/W)** One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the to address. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.14, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
41H	65	MSR_LASTBRANCH_1	**Last Branch Record 1 (R/W)** See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	**Last Branch Record 2 (R/W)** See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	**Last Branch Record 3 (R/W)** See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	**Last Branch Record 4 (R/W)** See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	**Last Branch Record 5 (R/W)** See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	**Last Branch Record 6 (R/W)** See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	**Last Branch Record 7 (R/W)** See description of MSR_LASTBRANCH_0.
119H	281	MSR_BBL_CR_CTL	
		63:0	Reserved.
11EH	281	MSR_BBL_CR_CTL3	
		0	**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		4:1	Reserved.
		5	**ECC Check Enable (RO)** This bit enables ECC checking on the cache data bus. ECC is always generated on write cycles. 0 = Disabled (default) 1 = Enabled For the Pentium M processor, ECC checking on the cache data bus is always enabled.
		7:6	Reserved.
		8	**L2 Enabled (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		22:9	Reserved.
		23	**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24	Reserved.
179H	377	IA32_MCG_CAP	
		7:0	**Count (RO)** Indicates the number of hardware unit error reporting banks available in the processor.
		8	**IA32_MCG_CTL Present (RO)** 1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH. 0 = Not supported.
		63:9	Reserved.
17AH	378	IA32_MCG_STATUS	
		0	**RIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1	**EIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2	**MCIP** When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3	Reserved.
198H	408	IA32_PERF_STATUS	See Table 2-2.
199H	409	IA32_PERF_CTL	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	**Clock Modulation (R/W).** See Table 2-2. See Section 14.7.3, “Software Controlled Clock Modulation.”
19BH	411	IA32_THERM_INTERRUPT	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, “Thermal Monitor.”
19CH	412	IA32_THERM_STATUS	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, “Thermal Monitor.”
19DH	413	MSR_THERM2_CTL	

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name	Bit Description
		15:0	Reserved.
		16	**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16	Reserved.
1A0H	416	IA32_MISC_ENABLE	**Enable Miscellaneous Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		2:0	Reserved.
		3	**Automatic Thermal Control Circuit Enable (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor's thermal sensor operation. 0 = Disabled (default). The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor's internal thermal sensor determines the processor is about to exceed its maximum operating temperature. When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature. The bit should not be confused with the on-demand thermal control circuit enable bit.
		6:4	Reserved.
		7	**Performance Monitoring Available (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled
		9:8	Reserved.
		10	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
			Branch Trace Storage Unavailable (RO) 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported
		12	**Processor Event Based Sampling Unavailable (RO)** 1 = Processor does not support processor event based sampling (PEBS); 0 = PEBS is supported. The Pentium M processor does not support PEBS.

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		15:13	Reserved.
		16	**Enhanced Intel SpeedStep Technology Enable (R/W)** 1 = Enhanced Intel SpeedStep Technology enabled. On the Pentium M processor, this bit may be configured to be read-only.
		22:17	Reserved.
		23	**xTPR Message Disable (R/W)** When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific.
		63:24	Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: ▪ MSR_LASTBRANCH_0_FROM_IP (at 40H) ▪ Section 17.14, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)”
1D9H	473	MSR_DEBUGCTLB	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.14, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).”
1DDH	477	MSR_LER_TO_LIP	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.14, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)” and Section 17.15.2, “Last Branch and Last Exception MSRs.”
1DEH	478	MSR_LER_FROM_LIP	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.14, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)” and Section 17.15.2, “Last Branch and Last Exception MSRs.”
2FFH	767	IA32_MTRR_DEF_TYPE	**Default Memory Types (R/W)** Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 11.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”
400H	1024	IA32_MC0_CTL	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”

Table 2-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
402H	1026	IA32_MC0_ADDR	See Section 14.3.2.3., “IA32_MCi_ADDR MSRs”. The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
405H	1029	IA32_MC1_STATUS	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
406H	1030	IA32_MC1_ADDR	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
409H	1033	IA32_MC2_STATUS	See Chapter 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40AH	1034	IA32_MC2_ADDR	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	MSR_MC4_STATUS	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
40EH	1038	MSR_MC4_ADDR	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	MSR_MC3_CTL	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
411H	1041	MSR_MC3_STATUS	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.”
412H	1042	MSR_MC3_ADDR	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
600H	1536	IA32_DS_AREA	**DS Save Area (R/W)** See Table 2-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.15.4, “Debug Store (DS) Mechanism.”
		31:0	**DS Buffer Management Area** Linear address of the first byte of the DS buffer management area.
		63:32	Reserved.

2.21 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

Table 2-46. MSRs in the P6 Family Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.16, "Time-Stamp Counter."
17H	23	IA32_PLATFORM_ID	**Platform ID (R)** The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
		49:0	Reserved.
		52:50	**Platform Id (R)** Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
		56:53	L2 Cache Latency Read.
		59:57	Reserved.
		60	Clock Frequency Ratio Read.
		63:61	Reserved.
1BH	27	APIC_BASE	Section 10.4.4, "Local APIC Status and Location."
		7:0	Reserved.
		8	**Boot Strap Processor indicator Bit** 1 = BSP
		10:9	Reserved.
		11	**APIC Global Enable Bit - Permanent till reset** 1 = Enabled 0 = Disabled
		31:12	APIC Base Address.
		63:32	Reserved.
2AH	42	EBL_CR_POWERON	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0	Reserved.[1]

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		1	**Data Error Checking Enable (R/W)** 1 = Enabled 0 = Disabled
		2	**Response Error Checking Enable FRCERR Observation Enable (R/W)** 1 = Enabled 0 = Disabled
		3	**AERR# Drive Enable (R/W)** 1 = Enabled 0 = Disabled
		4	**BERR# Enable for Initiator Bus Requests (R/W)** 1 = Enabled 0 = Disabled
		5	Reserved.
		6	**BERR# Driver Enable for Initiator Internal Errors (R/W)** 1 = Enabled 0 = Disabled
		7	**BINIT# Driver Enable (R/W)** 1 = Enabled 0 = Disabled
		8	**Output Tri-state Enabled (R)** 1 = Enabled 0 = Disabled
		9	**Execute BIST (R)** 1 = Enabled 0 = Disabled
		10	**AERR# Observation Enabled (R)** 1 = Enabled 0 = Disabled
		11	Reserved.
		12	**BINIT# Observation Enabled (R)** 1 = Enabled 0 = Disabled
		13	**In Order Queue Depth (R)** 1 = 1 0 = 8
		14	**1-MByte Power on Reset Vector (R)** 1 = 1MByte 0 = 4GBytes
		15	**FRC Mode Enable (R)** 1 = Enabled 0 = Disabled

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		17:16	**APIC Cluster ID (R)**
		19:18	**System Bus Frequency (R)** 00 = 66MHz 10 = 100Mhz 01 = 133MHz 11 = Reserved
		21: 20	**Symmetric Arbitration ID (R)**
		25:22	**Clock Frequency Ratio (R)**
		26	**Low Power Mode Enable (R/W)**
		27	**Clock Frequency Ratio**
		63:28	Reserved.[1]
33H	51	TEST_CTL	**Test Control Register**
		29:0	Reserved.
		30	**Streaming Buffer Disable**
		31	**Disable LOCK#** Assertion for split locked access.
79H	121	BIOS_UPDT_TRIG	BIOS Update Trigger Register.
88H	136	BBL_CR_D0[63:0]	Chunk 0 data register D[63:0]: used to write to and read from the L2
89H	137	BBL_CR_D1[63:0]	Chunk 1 data register D[63:0]: used to write to and read from the L2
8AH	138	BBL_CR_D2[63:0]	Chunk 2 data register D[63:0]: used to write to and read from the L2
8BH	139	BIOS_SIGN/BBL_CR_D3[63:0]	**BIOS Update Signature Register or Chunk 3 data register D[63:0]** Used to write to and read from the L2 depending on the usage model.
C1H	193	PerfCtr0 (PERFCTR0)	
C2H	194	PerfCtr1 (PERFCTR1)	
FEH	254	MTRRcap	
116H	278	BBL_CR_ADDR [63:0] BBL_CR_ADDR [63:32] BBL_CR_ADDR [31:3] BBL_CR_ADDR [2:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses. Reserved, Address bits [35:3] Reserved Set to 0.
118H	280	BBL_CR_DECC[63:0]	Data ECC register D[7:0]: used to write ECC and read ECC to/from L2
119H	281	BBL_CR_CTL BL_CR_CTL[63:22] BBL_CR_CTL[21]	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response Reserved Processor number[2] Disable = 1 Enable = 0 Reserved

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL[20:19]	User supplied ECC
		BBL_CR_CTL[18]	Reserved
		BBL_CR_CTL[17]	L2 Hit
		BBL_CR_CTL[16]	Reserved
		BBL_CR_CTL[15:14]	State from L2
		BBL_CR_CTL[13:12]	Modified - 11,Exclusive - 10, Shared - 01, Invalid - 00
			Way from L2
		BBL_CR_CTL[11:10]	Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11
			Way to L2
		BBL_CR_CTL[9:8]	Reserved
		BBL_CR_CTL[7]	State to L2
		BBL_CR_CTL[6:5]	
		BBL_CR_CTL[4:0] 01100 01110 01111 00010 00011 010 + MESI encode 111 + MESI encode 100 + MESI encode	L2 Command Data Read w/ LRU update (RLU) Tag Read w/ Data Read (TRR) Tag Inquire (TI) L2 Control Register Read (CR) L2 Control Register Write (CW) Tag Write w/ Data Read (TWR) Tag Write w/ Data Write (TWW) Tag Write (TW)
11AH	282	BBL_CR_TRIG	Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.
11BH	283	BBL_CR_BUSY	Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY
11EH	286	BBL_CR_CTL3	Control register 3: used to configure the L2 Cache
		BBL_CR_CTL3[63:26]	Reserved
		BBL_CR_CTL3[25]	Cache bus fraction (read only)
		BBL_CR_CTL3[24]	Reserved
		BBL_CR_CTL3[23]	L2 Hardware Disable (read only)
		BBL_CR_CTL3[22:20] 111 110 101 100 011 010 001 000	L2 Physical Address Range support 64GBytes 32GBytes 16GBytes 8GBytes 4GBytes 2GBytes 1GBytes 512MBytes
		BBL_CR_CTL3[19]	Reserved
		BBL_CR_CTL3[18]	Cache State error checking enable (read/write)

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13 00001 00010 00100 01000 10000	Cache size per bank (read/write) 256KBytes 512KBytes 1MByte 2MByte 4MBytes
		BBL_CR_CTL3[12:11]	Number of L2 banks (read only)
		BBL_CR_CTL3[10:9] 00 01 10 11	L2 Associativity (read only) Direct Mapped 2 Way 4 Way Reserved
		BBL_CR_CTL3[8]	L2 Enabled (read/write)
		BBL_CR_CTL3[7]	CRTN Parity Check Enable (read/write)
		BBL_CR_CTL3[6]	Address Parity Check Enable (read/write)
		BBL_CR_CTL3[5]	ECC Check Enable (read/write)
		BBL_CR_CTL3[4:1]	L2 Cache Latency (read/write)
		BBL_CR_CTL3[0]	L2 Configured (read/write)
174H	372	SYSENTER_CS_MSR	CS register target for CPL 0 code
175H	373	SYSENTER_ESP_MSR	Stack pointer for CPL 0 stack
176H	374	SYSENTER_EIP_MSR	CPL 0 code entry point
179H	377	MCG_CAP	
17AH	378	MCG_STATUS	
17BH	379	MCG_CTL	
186H	390	PerfEvtSel0 (EVNTSEL0)	
		7:0	**Event Select** Refer to Performance Counter section for a list of event encodings.
		15:8	**UMASK (Unit Mask)** Unit mask register set to 0 to enable all count options.
		16	**USER** Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	**OS** Controls the counting of events at Privilege level of 0.
		18	**E** Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	**PC** Enabled the signaling of performance counter overflow via BP0 pin

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address Hex	Register Address Dec	Register Name	Bit Description
		20	**INT** Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		22	**ENABLE** Enables the counting of performance events in both counters 1 = Enable 0 = Disable
		23	**INV** Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted
		31:24	CMASK (Counter Mask).
187H	391	PerfEvtSel1 (EVNTSEL1)	
		7:0	**Event Select** Refer to Performance Counter section for a list of event encodings.
		15:8	**UMASK (Unit Mask)** Unit mask register set to 0 to enable all count options.
		16	**USER** Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	**OS** Controls the counting of events at Privilege level of 0
		18	**E** Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	**PC** Enabled the signaling of performance counter overflow via BP0 pin.
		20	**INT** Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		23	**INV** Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		31:24	**CMASK (Counter Mask)**
1D9H	473	DEBUGCTLMSR	
		0	Enable/Disable Last Branch Records
		1	Branch Trap Flag
		2	Performance Monitoring/Break Point Pins
		3	Performance Monitoring/Break Point Pins
		4	Performance Monitoring/Break Point Pins
		5	Performance Monitoring/Break Point Pins
		6	Enable/Disable Execution Trace Messages
		31:7	Reserved
1DBH	475	LASTBRANCHFROMIP	
1DCH	476	LASTBRANCHTOIP	
1DDH	477	LASTINTFROMIP	
1DEH	478	LASTINTTOIP	
1E0H	480	ROB_CR_BKUPTMPDR6	
		1:0	Reserved
		2	Fast String Enable bit. Default is enabled
200H	512	MTRRphysBase0	
201H	513	MTRRphysMask0	
202H	514	MTRRphysBase1	
203H	515	MTRRphysMask1	
204H	516	MTRRphysBase2	
205H	517	MTRRphysMask2	
206H	518	MTRRphysBase3	
207H	519	MTRRphysMask3	
208H	520	MTRRphysBase4	
209H	521	MTRRphysMask4	
20AH	522	MTRRphysBase5	
20BH	523	MTRRphysMask5	
20CH	524	MTRRphysBase6	
20DH	525	MTRRphysMask6	
20EH	526	MTRRphysBase7	
20FH	527	MTRRphysMask7	
250H	592	MTRRfix64K_00000	
258H	600	MTRRfix16K_80000	
259H	601	MTRRfix16K_A0000	
268H	616	MTRRfix4K_C0000	

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
269H	617	MTRRfix4K_C8000	
26AH	618	MTRRfix4K_D0000	
26BH	619	MTRRfix4K_D8000	
26CH	620	MTRRfix4K_E0000	
26DH	621	MTRRfix4K_E8000	
26EH	622	MTRRfix4K_F0000	
26FH	623	MTRRfix4K_F8000	
2FFH	767	MTRRdefType	
		2:0	Default memory type
		10	Fixed MTRR enable
		11	MTRR Enable
400H	1024	MC0_CTL	
401H	1025	MC0_STATUS	
		15:0	MC_STATUS_MCACOD
		31:16	MC_STATUS_MSCOD
		57	MC_STATUS_DAM
		58	MC_STATUS_ADDRV
		59	MC_STATUS_MISCV
		60	MC_STATUS_EN. (Note: For MC0_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
402H	1026	MC0_ADDR	
403H	1027	MC0_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MC0_STATUS.
406H	1030	MC1_ADDR	
407H	1031	MC1_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
408H	1032	MC2_CTL	
409H	1033	MC2_STATUS	Bit definitions same as MC0_STATUS.
40AH	1034	MC2_ADDR	
40BH	1035	MC2_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
40CH	1036	MC4_CTL	
40DH	1037	MC4_STATUS	Bit definitions same as MC0_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.

Table 2-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
40EH	1038	MC4_ADDR	Defined in MCA architecture but not implemented in P6 Family processors.
40FH	1039	MC4_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
410H	1040	MC3_CTL	
411H	1041	MC3_STATUS	Bit definitions same as MC0_STATUS.
412H	1042	MC3_ADDR	
413H	1043	MC3_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

NOTES

1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL_CR_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL_CR_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

2.22 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5_MC_ADDR, P5_MC_TYPE, and TSC MSRs (named IA32_P5_MC_ADDR, IA32_P5_MC_TYPE, and IA32_TIME_STAMP_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

Table 2-47. MSRs in the Pentium Processor

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
1H	1	P5_MC_TYPE	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
10H	16	TSC	See Section 17.16, "Time-Stamp Counter."
11H	17	CESR	See Section 18.24.1, "Control and Event Select Register (CESR)."
12H	18	CTR0	Section 18.24.3, "Events Counted."
13H	19	CTR1	Section 18.24.3, "Events Counted."

2.23 MSR INDEX

MSRs of recent processors are indexed here for convenience. IA32 MSRs are excluded from this index.

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_ALF_ESCR0
0FH .... See Table 2-41
MSR_ALF_ESCR1
0FH .... See Table 2-41
MSR_ANY_CORE_C0
06_4EH, 06_5EH .... See Table 2-37
MSR_ANY_GFXE_C0
06_4EH, 06_5EH .... See Table 2-37
MSR_B0_PMON_BOX_CTRL
06_2EH .... See Table 2-15
MSR_B0_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_B0_PMON_BOX_STATUS
06_2EH .... See Table 2-15
MSR_B0_PMON_CTR0
06_2EH .... See Table 2-15
MSR_B0_PMON_CTR1
06_2EH .... See Table 2-15
MSR_B0_PMON_CTR2
06_2EH .... See Table 2-15
MSR_B0_PMON_CTR3
06_2EH .... See Table 2-15
MSR_B0_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
MSR_B0_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
MSR_B0_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
MSR_B0_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
MSR_B0_PMON_MASK
06_2EH .... See Table 2-15
MSR_B0_PMON_MATCH
06_2EH .... See Table 2-15
MSR_B1_PMON_BOX_CTRL
06_2EH .... See Table 2-15
MSR_B1_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_B1_PMON_BOX_STATUS
06_2EH .... See Table 2-15
MSR_B1_PMON_CTR0

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH .... See Table 2-15

MSR_B1_PMON_CTR1

06_2EH .... See Table 2-15

MSR_B1_PMON_CTR2

06_2EH .... See Table 2-15

MSR_B1_PMON_CTR3

06_2EH .... See Table 2-15

MSR_B1_PMON_EVNT_SEL0

06_2EH .... See Table 2-15

MSR_B1_PMON_EVNT_SEL1

06_2EH .... See Table 2-15

MSR_B1_PMON_EVNT_SEL2

06_2EH .... See Table 2-15

MSR_B1_PMON_EVNT_SEL3

06_2EH .... See Table 2-15

MSR_B1_PMON_MASK

06_2EH .... See Table 2-15

MSR_B1_PMON_MATCH

06_2EH .... See Table 2-15

MSR_BBL_CR_CTL

06_09H .... See Table 2-45

MSR_BBL_CR_CTL3

06_0FH, 06_17H .... See Table 2-3

06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .... See Table 2-4

06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-7

06_0EH .... See Table 2-44

06_09H .... See Table 2-45

MSR_BPU_CCCR0

0FH .... See Table 2-41

MSR_BPU_CCCR1

0FH .... See Table 2-41

MSR_BPU_CCCR2

0FH .... See Table 2-41

MSR_BPU_CCCR3

0FH .... See Table 2-41

MSR_BPU_COUNTER0

0FH .... See Table 2-41

MSR_BPU_COUNTER1

0FH .... See Table 2-41

MSR_BPU_COUNTER2

0FH .... See Table 2-41

MSR_BPU_COUNTER3

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

0FH .... See Table 2-41

MSR_BPU_ESCR0

0FH .... See Table 2-41

MSR_BPU_ESCR1

0FH .... See Table 2-41

MSR_BSU_ESCR0

0FH .... See Table 2-41

MSR_BSU_ESCR1

0FH .... See Table 2-41

MSR_C0_PMON_BOX_CTRL

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_BOX_FILTER

06_2DH .... See Table 2-22

MSR_C0_PMON_BOX_FILTER0

06_3FH .... See Table 2-31

MSR_C0_PMON_BOX_FILTER1

06_3EH .... See Table 2-26

06_3FH .... See Table 2-31

MSR_C0_PMON_BOX_OVF_CTRL

06_2EH .... See Table 2-15

MSR_C0_PMON_BOX_STATUS

06_2EH .... See Table 2-15

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR0

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR1

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR2

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR3

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR4

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH .... See Table 2-15

MSR_C0_PMON_CTR5

06_2EH .... See Table 2-15

MSR_C0_PMON_EVNT_SEL0

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_EVNT_SEL1

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR1

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_EVNT_SEL2

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_CTR2

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_EVNT_SEL3

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C0_PMON_EVNT_SEL4

06_2EH .... See Table 2-15

MSR_C0_PMON_EVNT_SEL5

06_2EH .... See Table 2-15

MSR_C1_PMON_BOX_CTRL

06_2EH .... See Table 2-15

06_2DH .... See Table 2-22

06_3FH .... See Table 2-31

MSR_C1_PMON_BOX_FILTER

06_2DH .... See Table 2-22

MSR_C1_PMON_BOX_FILTER0

06_3FH .... See Table 2-31

MSR_C1_PMON_BOX_FILTER1

06_3EH .... See Table 2-26

06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C1_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C1_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31
MSR_C1_PMON_CTR0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_CTR1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_CTR2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_CTR3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C1_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C1_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C1_PMON_EVNT_SEL4

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH .... See Table 2-15
MSR_C1_PMON_EVNT_SEL5
06_2EH .... See Table 2-15
MSR_C10_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C10_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C10_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C11_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C11_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C11_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C12_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C12_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C12_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C13_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C13_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C13_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C14_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C14_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C14_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C15_PMON_BOX_CTL
06_3FH .... See Table 2-31
MSR_C15_PMON_BOX_FILTER0
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C15_PMON_BOX_FILTER1
06_3FH .... See Table 2-31
MSR_C15_PMON_BOX_STATUS
06_3FH .... See Table 2-31
MSR_C15_PMON_CTR0
06_3FH .... See Table 2-31
MSR_C15_PMON_CTR1
06_3FH .... See Table 2-31
MSR_C15_PMON_CTR2
06_3FH .... See Table 2-31
MSR_C15_PMON_CTR3
06_3FH .... See Table 2-31
MSR_C15_PMON_EVNTSEL0
06_3FH .... See Table 2-31
MSR_C15_PMON_EVNTSEL1
06_3FH .... See Table 2-31
MSR_C15_PMON_EVNTSEL2
06_3FH .... See Table 2-31
MSR_C15_PMON_EVNTSEL3
06_3FH .... See Table 2-31
MSR_C16_PMON_BOX_CTL
06_3FH .... See Table 2-31
MSR_C16_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C16_PMON_BOX_FILTER1
06_3FH .... See Table 2-31
MSR_C16_PMON_BOX_STATUS
06_3FH .... See Table 2-31
MSR_C16_PMON_CTR0
06_3FH .... See Table 2-31
MSR_C16_PMON_CTR3
06_3FH .... See Table 2-31
MSR_C16_PMON_CTR2
06_3FH .... See Table 2-31
MSR_C16_PMON_CTR3
06_3FH .... See Table 2-31
MSR_C16_PMON_EVNTSEL0
06_3FH .... See Table 2-31
MSR_C16_PMON_EVNTSEL1
06_3FH .... See Table 2-31
MSR_C16_PMON_EVNTSEL2
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_C16_PMON_EVNTSEL3
06_3FH .... See Table 2-31
MSR_C17_PMON_BOX_CTL
06_3FH .... See Table 2-31
MSR_C17_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C17_PMON_BOX_FILTER1
06_3FH .... See Table 2-31
MSR_C17_PMON_BOX_STATUS
06_3FH .... See Table 2-31
MSR_C17_PMON_CTR0
06_3FH .... See Table 2-31
MSR_C17_PMON_CTR1
06_3FH .... See Table 2-31
MSR_C17_PMON_CTR2
06_3FH .... See Table 2-31
MSR_C17_PMON_CTR3
06_3FH .... See Table 2-31
MSR_C17_PMON_EVNTSEL0
06_3FH .... See Table 2-31
MSR_C17_PMON_EVNTSEL1
06_3FH .... See Table 2-31
MSR_C17_PMON_EVNTSEL2
06_3FH .... See Table 2-31
MSR_C17_PMON_EVNTSEL3
06_3FH .... See Table 2-31
MSR_C2_PMON_BOX_CTRL
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_BOX_FILTER
06_2DH .... See Table 2-22
MSR_C2_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C2_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C2_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C2_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C2_PMON_CTR0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_CTR1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_CTR2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_CTR3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C2_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C2_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C2_PMON_EVNT_SEL4
06_2EH .... See Table 2-15
MSR_C2_PMON_EVNT_SEL5
06_2EH .... See Table 2-15
MSR_C3_PMON_BOX_CTRL
06_2EH .... See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_BOX_FILTER
06_2DH .... See Table 2-22
MSR_C3_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C3_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C3_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C3_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31
MSR_C3_PMON_CTR0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_CTR1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_CTR2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_CTR3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C3_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C3_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C3_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C3_PMON_EVNT_SEL2
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C3_PMON_EVNT_SEL3
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C3_PMON_EVNT_SEL4
06_2EH . . . . See Table 2-15
MSR_C3_PMON_EVNT_SEL5
06_2EH . . . . See Table 2-15
MSR_C4_PMON_BOX_CTRL
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C4_PMON_BOX_FILTER
06_2DH . . . . See Table 2-22
MSR_C4_PMON_BOX_FILTER0
06_3FH . . . . See Table 2-31
MSR_C4_PMON_BOX_FILTER1
06_3EH . . . . See Table 2-26
06_3FH . . . . See Table 2-31
MSR_C4_PMON_BOX_OVF_CTRL
06_2EH . . . . See Table 2-15
MSR_C4_PMON_BOX_STATUS
06_2EH . . . . See Table 2-15
06_3FH . . . . See Table 2-31
MSR_C4_PMON_CTR0
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C4_PMON_CTR1
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C4_PMON_CTR2
06_2EH . . . . See Table 2-15
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31
MSR_C4_PMON_CTR3
06_2EH . . . . See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C4_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C4_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C4_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C4_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C4_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C4_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C4_PMON_EVNT_SEL4
06_2EH .... See Table 2-15
MSR_C4_PMON_EVNT_SEL5
06_2EH .... See Table 2-15
MSR_C5_PMON_BOX_CTRL
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C5_PMON_BOX_FILTER
06_2DH .... See Table 2-22
MSR_C5_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C5_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C5_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C5_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_C5_PMON_CTR0
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_CTR1
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_CTR2
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_CTR3
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_CTR4
06_2EH . . . . . . See Table 2-15
MSR_C5_PMON_CTR5
06_2EH . . . . . . See Table 2-15
MSR_C5_PMON_EVNT_SEL0
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_EVNT_SEL1
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_EVNT_SEL2
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_EVNT_SEL3
06_2EH . . . . . . See Table 2-15
06_2DH . . . . . . See Table 2-22
06_3FH . . . . . . See Table 2-31
MSR_C5_PMON_EVNT_SEL4
06_2EH . . . . . . See Table 2-15
MSR_C5_PMON_EVNT_SEL5
06_2EH . . . . . . See Table 2-15
MSR_C6_PMON_BOX_CTRL
06_2EH . . . . . . See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_BOX_FILTER
06_2DH .... See Table 2-22
MSR_C6_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C6_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C6_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C6_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31
MSR_C6_PMON_CTR0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_CTR1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_CTR2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_CTR3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C6_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C6_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_C6_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C6_PMON_EVNT_SEL4
06_2EH .... See Table 2-15
MSR_C6_PMON_EVNT_SEL5
06_2EH .... See Table 2-15
MSR_C7_PMON_BOX_CTRL
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_BOX_FILTER
06_2DH .... See Table 2-22
MSR_C7_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C7_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C7_PMON_BOX_OVF_CTRL
06_2EH .... See Table 2-15
MSR_C7_PMON_BOX_STATUS
06_2EH .... See Table 2-15
06_3FH .... See Table 2-31
MSR_C7_PMON_CTR0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_CTR1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_CTR2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_CTR3
06_2EH .... See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_CTR4
06_2EH .... See Table 2-15
MSR_C7_PMON_CTR5
06_2EH .... See Table 2-15
MSR_C7_PMON_EVNT_SEL0
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_EVNT_SEL1
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_EVNT_SEL2
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_EVNT_SEL3
06_2EH .... See Table 2-15
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_C7_PMON_EVNT_SEL4
06_2EH .... See Table 2-15
MSR_C7_PMON_EVNT_SEL5
06_2EH .... See Table 2-15
MSR_C8_PMON_BOX_CTRL
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C8_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C8_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_BOX_OVF_CTRL
06_2FH .... See Table 2-17
MSR_C8_PMON_BOX_STATUS
06_2FH .... See Table 2-17
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C8_PMON_CTR0
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_CTR1
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_CTR2
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_CTR3
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_CTR4
06_2FH .... See Table 2-17
MSR_C8_PMON_CTR5
06_2FH .... See Table 2-17
MSR_C8_PMON_EVNT_SEL0
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_EVNT_SEL1
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_EVNT_SEL2
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_EVNT_SEL3
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C8_PMON_EVNT_SEL4
06_2FH .... See Table 2-17
MSR_C8_PMON_EVNT_SEL5
06_2FH .... See Table 2-17
MSR_C9_PMON_BOX_CTRL
06_2FH .... See Table 2-17

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_BOX_FILTER
06_3EH .... See Table 2-26
MSR_C9_PMON_BOX_FILTER0
06_3FH .... See Table 2-31
MSR_C9_PMON_BOX_FILTER1
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_BOX_OVF_CTRL
06_2FH .... See Table 2-17
MSR_C9_PMON_BOX_STATUS
06_2FH .... See Table 2-17
06_3FH .... See Table 2-31
MSR_C9_PMON_CTR0
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_CTR1
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_CTR2
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_CTR3
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_CTR4
06_2FH .... See Table 2-17
MSR_C9_PMON_CTR5
06_2FH .... See Table 2-17
MSR_C9_PMON_EVNT_SEL0
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31
MSR_C9_PMON_EVNT_SEL1
06_2FH .... See Table 2-17
06_3EH .... See Table 2-26
06_3FH .... See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_C9_PMON_EVNT_SEL2
06_2FH . . . . . . See Table 2-17
06_3EH . . . . . . See Table 2-26
06_3FH . . . . . . See Table 2-31
MSR_C9_PMON_EVNT_SEL3
06_2FH . . . . . . See Table 2-17
06_3EH . . . . . . See Table 2-26
06_3FH . . . . . . See Table 2-31
MSR_C9_PMON_EVNT_SEL4
06_2FH . . . . . . See Table 2-17
MSR_C9_PMON_EVNT_SEL5
06_2FH . . . . . . See Table 2-17
MSR_CC6_DEMOTION_POLICY_CONFIG
06_37H . . . . . . See Table 2-9
MSR_CONFIG_TDP_CONTROL
06_3AH . . . . . . See Table 2-23
06_3CH, 06_45H, 06_46H . . . . . . See Table 2-27
06_57H . . . . . . See Table 2-40
MSR_CONFIG_TDP_LEVEL1
06_3AH . . . . . . See Table 2-23
06_3CH, 06_45H, 06_46H . . . . . . See Table 2-27
06_57H . . . . . . See Table 2-40
MSR_CONFIG_TDP_LEVEL2
06_3AH. . . . . . See Table 2-23
06_3CH, 06_45H, 06_46H . . . . . . See Table 2-27
06_57H . . . . . . See Table 2-40
MSR_CONFIG_TDP_NOMINAL
06_3AH . . . . . . See Table 2-23
06_3CH, 06_45H, 06_46H . . . . . . See Table 2-27
06_57H . . . . . . See Table 2-40
MSR_CORE_C1_RESIDENCY
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-6
MSR_CORE_C3_RESIDENCY
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH . . . . . . See Table 2-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . . . See Table 2-18
MSR_CORE_C6_RESIDENCY
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH . . . . . . See Table 2-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . . . See Table 2-18
06_57H . . . . . . See Table 2-40
MSR_CORE_C7_RESIDENCY

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .... See Table 2-18
MSR_CORE_GFXE_OVERLAP_C0
06_4EH, 06_5EH .... See Table 2-37
MSR_CORE_HDC_RESIDENCY
06_4EH, 06_5EH .... See Table 2-37
MSR_CORE_PERF_LIMIT_REASONS
06_5CH.. See Table 2-12
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_3F .... See Table 2-30
06_56H, 06_4FH .... See Table 2-34
06_57H .... See Table 2-40
MSR_CORE_THREAD_COUNT
06_3FH.. See Table 2-30
MSR_CRU_ESCR0
0FH .... See Table 2-41
MSR_CRU_ESCR1
0FH .... See Table 2-41
MSR_CRU_ESCR2
0FH .... See Table 2-41
MSR_CRU_ESCR3
0FH .... See Table 2-41
MSR_CRU_ESCR4
0FH .... See Table 2-41
MSR_CRU_ESCR5
0FH .... See Table 2-41
MSR_DAC_ESCR0
0FH .... See Table 2-41
MSR_DAC_ESCR1
0FH .... See Table 2-41
MSR_DRAM_ENERGY_ STATUS
06_5CH.. See Table 2-12
06_2DH .... See Table 2-21
06_3EH, 06_3FH .... See Table 2-24
06_3CH, 06_45H, 06_46H .... See Table 2-27
06_3F .... See Table 2-30
06_56H, 06_4FH .... See Table 2-34
06_57H .... See Table 2-40
MSR_DRAM_PERF_STATUS
06_5CH.. See Table 2-12
06_2DH .... See Table 2-21
06_3EH, 06_3FH .... See Table 2-24
06_3CH, 06_45H, 06_46H .... See Table 2-27

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3F .... See Table 2-30
06_56H, 06_4FH .... See Table 2-34
06_57H .... See Table 2-40
MSR_DRAM_POWER_INFO
06_5CH........ See Table 2-12
06_2DH .... See Table 2-21
06_3EH, 06_3FH .... See Table 2-24
06_3F .... See Table 2-30
06_56H, 06_4FH .... See Table 2-34
06_57H .... See Table 2-40
MSR_DRAM_POWER_LIMIT
06_5CH........ See Table 2-12
06_2DH .... See Table 2-21
06_3EH, 06_3FH .... See Table 2-24
06_3F .... See Table 2-30
06_56H, 06_4FH .... See Table 2-34
06_57H .... See Table 2-40
MSR_EBC_FREQUENCY_ID
0FH .... See Table 2-41
MSR_EBC_HARD_POWERON
0FH .... See Table 2-41
MSR_EBC_SOFT_POWERON
0FH .... See Table 2-41
MSR_EBL_CR_POWERON
06_0FH, 06_17H .... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_0EH .... See Table 2-44
06_09H .... See Table 2-45
MSR_EFSB_DRDY0
0F_03H, 0F_04H .... See Table 2-42
MSR_EFSB_DRDY1
0F_03H, 0F_04H .... See Table 2-42
MSR_EMON_L3_CTR_CTL0
06_0FH, 06_17H .... See Table 2-3
0F_06H .... See Table 2-43
MSR_EMON_L3_CTR_CTL1
06_0FH, 06_17H .... See Table 2-3
0F_06H .... See Table 2-43
MSR_EMON_L3_CTR_CTL2
06_0FH, 06_17H .... See Table 2-3
0F_06H .... See Table 2-43

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_EMON_L3_CTR_CTL3
06_0FH, 06_17H . . . . See Table 2-3
0F_06H . . . . See Table 2-43
MSR_EMON_L3_CTR_CTL4
06_0FH, 06_17H . . . . See Table 2-3
0F_06H . . . . See Table 2-43
MSR_EMON_L3_CTR_CTL5
06_0FH, 06_17H . . . . See Table 2-3
0F_06H . . . . See Table 2-43
MSR_EMON_L3_CTR_CTL6
06_0FH, 06_17H . . . . See Table 2-3
0F_06H . . . . See Table 2-43
MSR_EMON_L3_CTR_CTL7
06_0FH, 06_17H . . . . See Table 2-3
0F_06H . . . . See Table 2-43
MSR_EMON_L3_GL_CTL
06_0FH, 06_17H . . . . See Table 2-3
MSR_ERROR_CONTROL
06_2DH . . . . See Table 2-21
06_3EH . . . . See Table 2-24
06_3F . . . . See Table 2-30
MSR_FEATURE_CONFIG
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH. . . . . See Table 2-6
06_25H, 06_2CH. . . . . See Table 2-16
06_2FH . . . . See Table 2-17
06_2AH, 06_2DH. . . . . See Table 2-18
06_57H . . . . See Table 2-40
MSR_FIRM_ESCR0
0FH . . . . See Table 2-41
MSR_FIRM_ESCR1
0FH . . . . See Table 2-41
MSR_FLAME_CCCR0
0FH . . . . See Table 2-41
MSR_FLAME_CCCR1
0FH . . . . See Table 2-41
MSR_FLAME_CCCR2
0FH . . . . See Table 2-41
MSR_FLAME_CCCR3
0FH . . . . See Table 2-41
MSR_FLAME_COUNTER0
0FH . . . . See Table 2-41
MSR_FLAME_COUNTER1

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

0FH . . . . . . See Table 2-41
MSR_FLAME_COUNTER2
0FH . . . . . . See Table 2-41
MSR_FLAME_COUNTER3
0FH . . . . . . See Table 2-41
MSR_FLAME_ESCR0
0FH . . . . . . See Table 2-41
MSR_FLAME_ESCR1
0FH . . . . . . See Table 2-41
MSR_FSB_ESCR0
0FH . . . . . . See Table 2-41
MSR_FSB_ESCR1
0FH . . . . . . See Table 2-41
MSR_FSB_FREQ
06_0FH, 06_17H . . . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_4CH . . . . . . See Table 2-11
06_0EH . . . . . . See Table 2-44
MSR_GQ_SNOOP_MESF
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH . . . . . . See Table 2-14
MSR_GRAPHICS_PERF_LIMIT_REASONS
06_3CH, 06_45H, 06_46H . . . . . . See Table 2-28
MSR_IFSB_BUSQ0
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IFSB_BUSQ1
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IFSB_CNTR7
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IFSB_CTL6
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IFSB_SNPQ0
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IFSB_SNPQ1
0F_03H, 0F_04H . . . . . . See Table 2-42
MSR_IQ_CCCR0
0FH . . . . . . See Table 2-41
MSR_IQ_CCCR1
0FH . . . . . . See Table 2-41
MSR_IQ_CCCR2
0FH . . . . . . See Table 2-41
MSR_IQ_CCCR3

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
0FH . . . . . See Table 2-41
MSR_IQ_CCCR4
0FH . . . . . See Table 2-41
MSR_IQ_CCCR5
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER0
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER1
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER2
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER3
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER4
0FH . . . . . See Table 2-41
MSR_IQ_COUNTER5
0FH . . . . . See Table 2-41
MSR_IQ_ESCR0
0FH . . . . . See Table 2-41
MSR_IQ_ESCR1
0FH . . . . . See Table 2-41
MSR_IS_ESCR0
0FH . . . . . See Table 2-41
MSR_IS_ESCR1
0FH . . . . . See Table 2-41
MSR_ITLB_ESCR0
0FH . . . . . See Table 2-41
MSR_ITLB_ESCR1
0FH . . . . . See Table 2-41
MSR_IX_ESCR0
0FH . . . . . See Table 2-41
MSR_IX_ESCR1
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_0
0FH . . . . . See Table 2-41
06_0EH . . . . . See Table 2-44
06_09H . . . . . See Table 2-45
MSR_LASTBRANCH_0_FROM_IP
06_0FH, 06_17H . . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . See Table 2-7
06_5CH. . . . . See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_0_TO_IP
06_0FH, 06_17H . . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_1_FROM_IP
06_0FH, 06_17H . . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_1_TO_IP
06_0FH, 06_17H . . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_10_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_10_TO_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41
MSR_LASTBRANCH_11_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . See Table 2-18
0FH . . . . . See Table 2-41

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_LASTBRANCH_11_TO_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_12_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_12_TO_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_13_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_13_TO_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_14_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_14_TO_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_15_FROM_IP
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH . . . . See Table 2-41
MSR_LASTBRANCH_15_TO_IP
06_5CH. . . . . See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2AH, 06_2DH .... See Table 2-18
0FH........ See Table 2-41
MSR_LASTBRANCH_16_FROM_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_16_TO_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_17_FROM_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_17_TO_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_18_FROM_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_18_TO_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_19_FROM_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_19_TO_IP
06_5CH........ See Table 2-12
06_4EH, 06_5EH........ See Table 2-37
MSR_LASTBRANCH_2
0FH........ See Table 2-41
06_0EH........ See Table 2-44
06_09H........ See Table 2-45
MSR_LASTBRANCH_2_FROM_IP
06_0FH, 06_17H........ See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H........ See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH........ See Table 2-7
06_5CH........ See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH........ See Table 2-13
06_2AH, 06_2DH........ See Table 2-18
0FH........ See Table 2-41
MSR_LASTBRANCH_2_TO_IP
06_0FH, 06_17H........ See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H........ See Table 2-4

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH. . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH. . . . . See Table 2-13
06_2AH, 06_2DH. . . . . See Table 2-18
0FH. . . . . See Table 2-41
MSR_LASTBRANCH_20_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_LASTBRANCH_20_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_LASTBRANCH_21_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_LASTBRANCH_21_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_22_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_22_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_23_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_23_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_24_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_24_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_25_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_25_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_26_FROM_IP

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_26_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_27_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_27_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_28_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_28_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_29_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_29_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_3
0FH. . . . . See Table 2-41
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_LASTBRANCH_3_FROM_IP
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH. . . . . See Table 2-41
MSR_LASTBRANCH_3_TO_IP
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

0FH. . . . . See Table 2-41
MSR_LASTBRANCH_30_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_30_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_31_FROM_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_31_TO_IP
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LASTBRANCH_4
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_LASTBRANCH_4_FROM_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH. . . . . See Table 2-41
MSR_LASTBRANCH_4_TO_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH. . . . . See Table 2-41
MSR_LASTBRANCH_5
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_LASTBRANCH_5_FROM_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
0FH. . . . . See Table 2-41
MSR_LASTBRANCH_5_TO_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . . See Table 2-18
0FH. . . . . . See Table 2-41
MSR_LASTBRANCH_6
06_0EH. . . . . . See Table 2-44
06_09H. . . . . . See Table 2-45
MSR_LASTBRANCH_6_FROM_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . . See Table 2-18
0FH. . . . . . See Table 2-41
MSR_LASTBRANCH_6_TO_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . . See Table 2-18
0FH. . . . . . See Table 2-41
MSR_LASTBRANCH_7
06_0EH. . . . . . See Table 2-44
06_09H. . . . . . See Table 2-45
MSR_LASTBRANCH_7_FROM_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . . See Table 2-18
0FH. . . . . . See Table 2-41
MSR_LASTBRANCH_7_TO_IP
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . . . See Table 2-7
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13
06_2AH, 06_2DH . . . . . . See Table 2-18
0FH. . . . . . See Table 2-41
MSR_LASTBRANCH_8_FROM_IP
06_5CH. . . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . . . See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2AH, 06_2DH ... See Table 2-18
0FH... See Table 2-41
MSR_LASTBRANCH_8_TO_IP
06_5CH... See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH ... See Table 2-13
06_2AH, 06_2DH... See Table 2-18
0FH... See Table 2-41
MSR_LASTBRANCH_9_FROM_IP
06_5CH... See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH ... See Table 2-13
06_2AH, 06_2DH ... See Table 2-18
0FH... See Table 2-41
MSR_LASTBRANCH_9_TO_IP
06_5CH... See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH ... See Table 2-13
06_2AH, 06_2DH ... See Table 2-18
0FH... See Table 2-41
MSR_LASTBRANCH_TOS
06_0FH, 06_17H ... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H ... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH ... See Table 2-7
06_5CH... See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH ... See Table 2-13
06_2AH, 06_2DH ... See Table 2-18
06_4EH, 06_5EH ... See Table 2-37
06_57H... See Table 2-40
06_0EH... See Table 2-44
06_09H... See Table 2-45
MSR_LBR_INFO_1
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_10
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_11
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_12
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_13
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_14
06_4EH, 06_5EH ... See Table 2-37
MSR_LBR_INFO_15
06_4EH, 06_5EH ... See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_LBR_INFO_16
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_17
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_18
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_19
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_2
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_20
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_21
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_22
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_23
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_24
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_25
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_26
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_27
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_28
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_29
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_3
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_30
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_31
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_4
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_5
06_4EH, 06_5EH .... See Table 2-37
MSR_LBR_INFO_6
06_4EH, 06_5EH .... See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_LBR_INFO_7
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LBR_INFO_8
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LBR_INFO_9
06_4EH, 06_5EH . . . . See Table 2-37
MSR_LBR_SELECT
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_3CH, 06_45H, 06_46H . . . . See Table 2-27
06_57H . . . . See Table 2-40
MSR_LER_FROM_LIP
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_57H. . . . . See Table 2-40
0FH. . . . . See Table 2-41
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_LER_TO_LIP
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_57H. . . . . See Table 2-40
0FH. . . . . See Table 2-41
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_M0_PMON_ADDR_MASK
06_2EH. . . . . See Table 2-15
MSR_M0_PMON_ADDR_MATCH
06_2EH. . . . . See Table 2-15
MSR_M0_PMON_BOX_CTRL
06_2EH. . . . . See Table 2-15
MSR_M0_PMON_BOX_OVF_CTRL
06_2EH. . . . . See Table 2-15
MSR_M0_PMON_BOX_STATUS

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH.. See Table 2-15
MSR_M0_PMON_CTR0
06_2EH.. See Table 2-15
MSR_M0_PMON_CTR1
06_2EH.. See Table 2-15
MSR_M0_PMON_CTR2
06_2EH.. See Table 2-15
MSR_M0_PMON_CTR3
06_2EH.. See Table 2-15
MSR_M0_PMON_CTR4
06_2EH.. See Table 2-15
MSR_M0_PMON_CTR5
06_2EH.. See Table 2-15
MSR_M0_PMON_DSP
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL0
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL1
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL2
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL3
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL4
06_2EH.. See Table 2-15
MSR_M0_PMON_EVNT_SEL5
06_2EH.. See Table 2-15
MSR_M0_PMON_ISS
06_2EH.. See Table 2-15
MSR_M0_PMON_MAP
06_2EH.. See Table 2-15
MSR_M0_PMON_MM_CONFIG
06_2EH.. See Table 2-15
MSR_M0_PMON_MSC_THR
06_2EH.. See Table 2-15
MSR_M0_PMON_PGT
06_2EH.. See Table 2-15
MSR_M0_PMON_PLD
06_2EH.. See Table 2-15
MSR_M0_PMON_TIMESTAMP
06_2EH.. See Table 2-15
MSR_M0_PMON_ZDP

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH.......... See Table 2-15

MSR_M1_PMON_ADDR_MASK

06_2EH.......... See Table 2-15

MSR_M1_PMON_ADDR_MATCH

06_2EH.......... See Table 2-15

MSR_M1_PMON_BOX_CTRL

06_2EH.......... See Table 2-15

MSR_M1_PMON_BOX_OVF_CTRL

06_2EH.......... See Table 2-15

MSR_M1_PMON_BOX_STATUS

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR0

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR1

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR2

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR3

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR4

06_2EH.......... See Table 2-15

MSR_M1_PMON_CTR5

06_2EH.......... See Table 2-15

MSR_M1_PMON_DSP

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL0

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL1

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL2

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL3

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL4

06_2EH.......... See Table 2-15

MSR_M1_PMON_EVNT_SEL5

06_2EH.......... See Table 2-15

MSR_M1_PMON_ISS

06_2EH.......... See Table 2-15

MSR_M1_PMON_MAP

06_2EH.......... See Table 2-15

MSR_M1_PMON_MM_CONFIG

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2EH. . . . . See Table 2-15
MSR_M1_PMON_MSC_THR
06_2EH. . . . . See Table 2-15
MSR_M1_PMON_PGT
06_2EH. . . . . See Table 2-15
MSR_M1_PMON_PLD
06_2EH. . . . . See Table 2-15
MSR_M1_PMON_TIMESTAMP
06_2EH. . . . . See Table 2-15
MSR_M1_PMON_ZDP
06_2EH. . . . . See Table 2-15
IA32_MC0_MISC / MSR_MC0_MISC
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
MSR_MC0_RESIDENCY
06_57H. . . . . See Table 2-40
IA32_MC1_MISC / MSR_MC1_MISC
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
IA32_MC10_ADDR / MSR_MC10_ADDR
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . . See Table 2-36
IA32_MC10_CTL / MSR_MC10_CTL
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . . See Table 2-36
IA32_MC10_MISC / MSR_MC10_MISC
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . . See Table 2-36
IA32_MC10_STATUS / MSR_MC10_STATUS
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3F. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . . See Table 2-36
IA32_MC11_ADDR / MSR_MC11_ADDR
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC11_CTL / MSR_MC11_CTL
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC11_MISC / MSR_MC11_MISC
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC11_STATUS / MSR_MC11_STATUS
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC12_ADDR / MSR_MC12_ADDR
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC12_CTL / MSR_MC12_CTL
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36
IA32_MC12_MISC / MSR_MC12_MISC
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC12_STATUS / MSR_MC12_STATUS
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC13_ADDR / MSR_MC13_ADDR
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC13_CTL / MSR_MC13_CTL
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC13_MISC / MSR_MC13_MISC
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC13_STATUS / MSR_MC13_STATUS
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC14_ADDR / MSR_MC14_ADDR
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4FH. . . . . See Table 2-36

IA32_MC14_CTL / MSR_MC14_CTL
06_2EH. . . . . See Table 2-15
06_2DH. . . . . See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC14_MISC / MSR_MC14_MISC
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC14_STATUS / MSR_MC14_STATUS
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC15_ADDR / MSR_MC15_ADDR
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC15_CTL / MSR_MC15_CTL
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC15_MISC / MSR_MC15_MISC
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC15_STATUS / MSR_MC15_STATUS
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_4FH........ See Table 2-36
IA32_MC16_ADDR / MSR_MC16_ADDR
06_2EH........ See Table 2-15
06_2DH........ See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_4FH. . . . See Table 2-36
IA32_MC16_CTL / MSR_MC16_CTL
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_4FH. . . . See Table 2-36
IA32_MC16_MISC / MSR_MC16_MISC
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_4FH. . . . See Table 2-36
IA32_MC16_STATUS / MSR_MC16_STATUS
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_4FH. . . . See Table 2-36
IA32_MC17_ADDR / MSR_MC17_ADDR
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . See Table 2-36
IA32_MC17_CTL / MSR_MC17_CTL
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . See Table 2-36
IA32_MC17_MISC / MSR_MC17_MISC
06_2EH. . . . See Table 2-15
06_2DH. . . . See Table 2-21
06_3EH. . . . See Table 2-24
06_3F. . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. . . . See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
IA32_MC17_STATUS / MSR_MC17_STATUS
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36
IA32_MC18_ADDR / MSR_MC18_ADDR
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36
IA32_MC18_CTL / MSR_MC18_CTL
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36
IA32_MC18_MISC / MSR_MC18_MISC
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36
IA32_MC18_STATUS / MSR_MC18_STATUS
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36
IA32_MC19_ADDR / MSR_MC19_ADDR
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

IA32_MC19_CTL / MSR_MC19_CTL
06_2EH.... See Table 2-15
06_2DH.... See Table 2-21
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH.... See Table 2-36
IA32_MC19_MISC / MSR_MC19_MISC
06_2EH.... See Table 2-15
06_2DH.... See Table 2-21
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH.... See Table 2-36
IA32_MC19_STATUS / MSR_MC19_STATUS
06_2EH.... See Table 2-15
06_2DH.... See Table 2-21
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH.... See Table 2-36
IA32_MC2_MISC / MSR_MC2_MISC
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
IA32_MC20_ADDR / MSR_MC20_ADDR
06_2EH.... See Table 2-15
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_4FH.... See Table 2-36
IA32_MC20_CTL / MSR_MC20_CTL
06_2EH.... See Table 2-15
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_4FH.... See Table 2-36
IA32_MC20_MISC / MSR_MC20_MISC
06_2EH.... See Table 2-15
06_3EH.... See Table 2-24
06_3F.... See Table 2-30
06_4FH.... See Table 2-36
IA32_MC20_STATUS / MSR_MC20_STATUS
06_2EH.... See Table 2-15
06_3EH.... See Table 2-24
06_3F.... See Table 2-30

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_4FH. . . . . See Table 2-36
IA32_MC21_ADDR / MSR_MC21_ADDR
06_2EH. . . . . See Table 2-15
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4F. . . . . See Table 2-36
IA32_MC21_CTL / MSR_MC21_CTL
06_2EH. . . . . See Table 2-15
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4F. . . . . See Table 2-36
IA32_MC21_MISC / MSR_MC21_MISC
06_2EH. . . . . See Table 2-15
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4F. . . . . See Table 2-36
IA32_MC21_STATUS / MSR_MC21_STATUS
06_2EH. . . . . See Table 2-15
06_3EH. . . . . See Table 2-24
06_3F. . . . . See Table 2-30
06_4F. . . . . See Table 2-36
IA32_MC22_ADDR / MSR_MC22_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC22_CTL / MSR_MC22_CTL
06_3EH. . . . . See Table 2-24
IA32_MC22_MISC / MSR_MC22_MISC
06_3EH. . . . . See Table 2-24
IA32_MC22_STATUS / MSR_MC22_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC23_ADDR / MSR_MC23_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC23_CTL / MSR_MC23_CTL
06_3EH. . . . . See Table 2-24
IA32_MC23_MISC / MSR_MC23_MISC
06_3EH. . . . . See Table 2-24
IA32_MC23_STATUS / MSR_MC23_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC24_ADDR / MSR_MC24_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC24_CTL / MSR_MC24_CTL
06_3EH. . . . . See Table 2-24
IA32_MC24_MISC / MSR_MC24_MISC

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3EH. . . . . See Table 2-24
IA32_MC24_STATUS / MSR_MC24_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC25_ADDR / MSR_MC25_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC25_CTL / MSR_MC25_CTL
06_3EH. . . . . See Table 2-24
IA32_MC25_MISC / MSR_MC25_MISC
06_3EH. . . . . See Table 2-24
IA32_MC25_STATUS / MSR_MC25_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC26_ADDR / MSR_MC26_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC26_CTL / MSR_MC26_CTL
06_3EH. . . . . See Table 2-24
IA32_MC26_MISC / MSR_MC26_MISC
06_3EH. . . . . See Table 2-24
IA32_MC26_STATUS / MSR_MC26_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC27_ADDR / MSR_MC27_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC27_CTL / MSR_MC27_CTL
06_3EH. . . . . See Table 2-24
IA32_MC27_MISC / MSR_MC27_MISC
06_3EH. . . . . See Table 2-24
IA32_MC27_STATUS / MSR_MC27_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC28_ADDR / MSR_MC28_ADDR
06_3EH. . . . . See Table 2-24
IA32_MC28_CTL / MSR_MC28_CTL
06_3EH. . . . . See Table 2-24
IA32_MC28_MISC / MSR_MC28_MISC
06_3EH. . . . . See Table 2-24
IA32_MC28_STATUS / MSR_MC28_STATUS
06_3EH. . . . . See Table 2-24
IA32_MC29_ADDR / MSR_MC29_ADDR
06_3EH. . . . . See Table 2-25
IA32_MC29_CTL / MSR_MC29_CTL
06_3EH. . . . . See Table 2-25
IA32_MC29_MISC / MSR_MC29_MISC
06_3EH. . . . . See Table 2-25
IA32_MC29_STATUS / MSR_MC29_STATUS

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3EH.......... See Table 2-25
IA32_MC3_ADDR / MSR_MC3_ADDR
06_0FH, 06_17H.......... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.......... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.......... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_57H.......... See Table 2-40
06_0EH.......... See Table 2-44
06_09H.......... See Table 2-45
IA32_MC3_CTL / MSR_MC3_CTL
06_0FH, 06_17H.......... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.......... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.......... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_57H.......... See Table 2-40
06_0EH.......... See Table 2-44
06_09H.......... See Table 2-45
IA32_MC3_MISC / MSR_MC3_MISC
06_0FH, 06_17H.......... See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_0EH.......... See Table 2-44
IA32_MC3_STATUS / MSR_MC3_STATUS
06_0FH, 06_17H.......... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.......... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.......... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_57H.......... See Table 2-40
06_0EH.......... See Table 2-44
06_09H.......... See Table 2-45
IA32_MC30_ADDR / MSR_MC30_ADDR
06_3EH.......... See Table 2-25
IA32_MC30_CTL / MSR_MC30_CTL
06_3EH.......... See Table 2-25
IA32_MC30_MISC / MSR_MC30_MISC
06_3EH.......... See Table 2-25
IA32_MC30_STATUS / MSR_MC30_STATUS
06_3EH.......... See Table 2-25
IA32_MC31_ADDR / MSR_MC31_ADDR
06_3EH.......... See Table 2-25
IA32_MC31_CTL / MSR_MC31_CTL
06_3EH.......... See Table 2-25
IA32_MC31_MISC / MSR_MC31_MISC

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3EH .... See Table 2-25
IA32_MC31_STATUS / MSR_MC31_STATUS
06_3EH .... See Table 2-25
IA32_MC4_ADDR / MSR_MC4_ADDR
06_0FH, 06_17H .... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_57H .... See Table 2-40
06_0EH .... See Table 2-44
06_09H .... See Table 2-45
IA32_MC4_CTL / MSR_MC4_CTL
06_0FH, 06_17H .... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_57H .... See Table 2-40
06_0EH .... See Table 2-44
06_09H .... See Table 2-45
IA32_MC4_CTL2 / MSR_MC4_CTL2
06_2AH, 06_2DH .... See Table 2-18
IA32_MC4_STATUS / MSR_MC4_STATUS
06_0FH, 06_17H .... See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .... See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_57H .... See Table 2-40
06_0EH .... See Table 2-44
06_09H .... See Table 2-45
MSR_MC5_ADDR / MSR_MC5_ADDR
06_0FH, 06_17H .... See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH .... See Table 2-21
06_3EH .... See Table 2-24
06_3FH .... See Table 2-30
06_4FH .... See Table 2-36
06_57H .... See Table 2-40
06_0EH .... See Table 2-44
IA32_MC5_CTL / MSR_MC5_CTL
06_0FH, 06_17H .... See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3FH........ See Table 2-30
06_4FH........ See Table 2-36
06_57H........ See Table 2-40
06_0EH........ See Table 2-44
IA32_MC5_MISC / MSR_MC5_MISC
06_0FH, 06_17H .... See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3FH........ See Table 2-30
06_4FH........ See Table 2-36
06_0EH........ See Table 2-44
IA32_MC5_STATUS / MSR_MC5_STATUS
06_0FH, 06_17H .... See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3FH........ See Table 2-30
06_4FH........ See Table 2-36
06_57H........ See Table 2-40
06_0EH........ See Table 2-44
IA32_MC6_ADDR / MSR_MC6_ADDR
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
IA32_MC6_CTL / MSR_MC6_CTL
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
MSR_MC6_DEMOTION_POLICY_CONFIG
06_37H........ See Table 2-9
IA32_MC6_MISC / MSR_MC6_MISC

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
MSR_MC6_RESIDENCY_COUNTER
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-7
06_37H........ See Table 2-9
06_57H........ See Table 2-40
IA32_MC6_STATUS / MSR_MC6_STATUS
06_0FH, 06_17H .... See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3FH........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
IA32_MC7_ADDR / MSR_MC7_ADDR
06_1AH, 06_1EH, 06_1FH, 06_2EH See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
IA32_MC7_CTL / MSR_MC7_CTL
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
IA32_MC7_MISC / MSR_MC7_MISC
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21
06_3EH........ See Table 2-24
06_3F........ See Table 2-30
06_56H, 06_4FH .... See Table 2-35
06_4FH........ See Table 2-36
IA32_MC7_STATUS / MSR_MC7_STATUS
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2DH........ See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36

IA32_MC8_ADDR / MSR_MC8_ADDR
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_4FH.......... See Table 2-36

IA32_MC8_CTL / MSR_MC8_CTL
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_4FH.......... See Table 2-36

IA32_MC8_MISC / MSR_MC8_MISC
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_4FH.......... See Table 2-36

IA32_MC8_STATUS / MSR_MC8_STATUS
06_1AH, 06_1EH, 06_1FH, 06_2EH.......... See Table 2-13
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_4FH.......... See Table 2-36

IA32_MC9_ADDR / MSR_MC9_ADDR
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36

IA32_MC9_CTL / MSR_MC9_CTL
06_2EH.......... See Table 2-15
06_2DH.......... See Table 2-21
06_3EH.......... See Table 2-24
06_3F.......... See Table 2-30
06_56H, 06_4FH.......... See Table 2-35
06_4FH.......... See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
IA32_MC9_MISC / MSR_MC9_MISC
06_2EH. See Table 2-15
06_2DH. See Table 2-21
06_3EH. See Table 2-24
06_3F. See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. See Table 2-36
IA32_MC9_STATUS / MSR_MC9_STATUS
06_2EH. See Table 2-15
06_2DH. See Table 2-21
06_3EH. See Table 2-24
06_3F. See Table 2-30
06_56H, 06_4FH . . . . See Table 2-35
06_4FH. See Table 2-36
MSR_MCG_MISC
0FH. See Table 2-41
MSR_MCG_R10
0FH. See Table 2-41
MSR_MCG_R11
0FH. See Table 2-41
MSR_MCG_R12
0FH. See Table 2-41
MSR_MCG_R13
0FH. See Table 2-41
MSR_MCG_R14
0FH. See Table 2-41
MSR_MCG_R15
0FH. See Table 2-41
MSR_MCG_R8
0FH. See Table 2-41
MSR_MCG_R9
0FH. See Table 2-41
MSR_MCG_RAX
0FH. See Table 2-41
MSR_MCG_RBP
0FH. See Table 2-41
MSR_MCG_RBX
0FH. See Table 2-41
MSR_MCG_RCX
0FH. See Table 2-41
MSR_MCG_RDI
0FH. See Table 2-41

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_MCG_RDX
0FH. . . . . See Table 2-41
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5
0FH. . . . . See Table 2-41
MSR_MCG_RFLAGS
0FH. . . . . See Table 2-41
MSR_MCG_RIP
0FH. . . . . See Table 2-41
MSR_MCG_RSI
0FH. . . . . See Table 2-41
MSR_MCG_RSP
0FH. . . . . See Table 2-41
MSR_MISC_FEATURE_CONTROL
06_5CH . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
MSR_MISC_PWR_MGMT
06_5CH . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
MSR_MOB_ESCR0
0FH. . . . . See Table 2-41
MSR_MOB_ESCR1
0FH. . . . . See Table 2-41
MSR_MS_CCCR0
0FH. . . . . See Table 2-41
MSR_MS_CCCR1
0FH. . . . . See Table 2-41
MSR_MS_CCCR2
0FH. . . . . See Table 2-41
MSR_MS_CCCR3
0FH. . . . . See Table 2-41
MSR_MS_COUNTER0
0FH. . . . . See Table 2-41
MSR_MS_COUNTER1
0FH. . . . . See Table 2-41
MSR_MS_COUNTER2
0FH. . . . . See Table 2-41
MSR_MS_COUNTER3
0FH. . . . . See Table 2-41
MSR_MS_ESCR0
0FH. . . . . See Table 2-41

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_MS_ESCR1
0FH. . . . . See Table 2-41
MSR_MTRRCAP
06_4EH, 06_5EH . . . . See Table 2-37
MSR_OFFCORE_RSP_0
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_57H. . . . . See Table 2-40
MSR_OFFCORE_RSP_1
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_25H, 06_2CH . . . . See Table 2-16
06_2FH. . . . . See Table 2-17
06_2AH, 06_2DH . . . . See Table 2-18
06_57H. . . . . See Table 2-40
MSR_PCIE_PLL_RATIO
06_3FH. . . . . See Table 2-30
MSR_PCU_PMON_BOX_CTL
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_BOX_FILTER
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_BOX_STATUS
06_3EH. . . . . See Table 2-26
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_CTR0
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_CTR1
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_CTR2
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_CTR3
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_EVNTSEL0
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_EVNTSEL1

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_EVNTSEL2
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PCU_PMON_EVNTSEL3
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31
MSR_PEBS_ENABLE
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_3EH. . . . . See Table 2-25
06_57H. . . . . See Table 2-40
0FH. . . . . See Table 2-41
MSR_PEBS_FRONTEND
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_PEBS_LD_LAT
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
MSR_PEBS_MATRIX_VERT
0FH. . . . . See Table 2-41
MSR_PEBS_NUM_ALT
06_2DH. . . . . See Table 2-21
MSR_PERF_CAPABILITIES
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_FIXED_CTR_CTRL
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_FIXED_CTR0
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_FIXED_CTR1
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_FIXED_CTR2
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_GLOBAL_CTRL
06_0FH, 06_17H . . . . See Table 2-3
MSR_PERF_GLOBAL_OVF_CTRL
06_0FH, 06_17H . . . . See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_PERF_GLOBAL_STATUS

06_0FH, 06_17H . . . . See Table 2-3

06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13

MSR_PERF_STATUS

06_0FH, 06_17H . . . . See Table 2-3

06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4

06_2AH, 06_2DH . . . . See Table 2-18

MSR_PKG_C10_RESIDENCY

06_5CH. . . . . See Table 2-12

06_45H. . . . . See Table 2-28 and Table 2-29

06_4FH. . . . . See Table 2-36

MSR_PKG_C2_RESIDENCY

06_27H. . . . . See Table 2-5

06_5CH. . . . . See Table 2-12

06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18

06_57H. . . . . See Table 2-40

MSR_PKG_C3_RESIDENCY

06_5CH. . . . . See Table 2-12

06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH . . . . See Table 2-13

06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18

06_57H. . . . . See Table 2-40

MSR_PKG_C4_RESIDENCY

06_27H. . . . . See Table 2-5

MSR_PKG_C6_RESIDENCY

06_27H. . . . . See Table 2-5

06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7

06_5CH. . . . . See Table 2-12

06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH . . . . See Table 2-13

06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18

06_57H. . . . . See Table 2-40

MSR_PKG_C7_RESIDENCY

06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH . . . . See Table 2-13

06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18

06_57H. . . . . See Table 2-40

MSR_PKG_C8_RESIDENCY

06_45H. . . . . See Table 2-29

06_4FH. . . . . See Table 2-36

MSR_PKG_C9_RESIDENCY

06_45H. . . . . See Table 2-29

06_4FH. . . . . See Table 2-36

MSR_PKG_CST_CONFIG_CONTROL

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .... See Table 2-7
06_4CH.......... See Table 2-11
06_5CH.......... See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .... See Table 2-13
06_2AH, 06_2DH .... See Table 2-18
06_3AH.......... See Table 2-23
06_3EH.......... See Table 2-24
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_45H.......... See Table 2-29
06_3F.......... See Table 2-30
06_3DH.......... See Table 2-33
06_56H, 06_4FH .... See Table 2-34
06_57H.......... See Table 2-40
MSR_PKG_ENERGY_STATUS
06_37H, 06_4AH, 06_5AH, 06_5DH .... See Table 2-8
06_5CH.......... See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .... See Table 2-18
MSR_PKG_HDC_CONFIG
06_4EH, 06_5EH.......... See Table 2-37
MSR_PKG_HDC_DEEP_RESIDENCY
06_4EH, 06_5EH .... See Table 2-37
MSR_PKG_HDC_SHALLOW_RESIDENCY
06_4EH, 06_5EH .... See Table 2-37
MSR_PKG_PERF_STATUS
06_5CH.......... See Table 2-12
06_2DH.......... See Table 2-21
06_3EH, 06_3FH .... See Table 2-24
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_57H.......... See Table 2-40
MSR_PKG_POWER_INFO
06_4DH.......... See Table 2-10
06_5CH.......... See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .... See Table 2-18
06_57H.......... See Table 2-40
MSR_PKG_POWER_LIMIT
06_37H, 06_4AH, 06_5AH, 06_5DH .... See Table 2-8
06_4DH.......... See Table 2-10
06_5CH.......... See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .... See Table 2-18
06_57H.......... See Table 2-40
MSR_PKGC_IRTL1
06_5CH.......... See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3CH, 06_45H, 06_46H. . . . . See Table 2-27
MSR_PKGC_IRTL2
06_5CH. . . . . See Table 2-12
06_3CH, 06_45H, 06_46H. . . . . See Table 2-27
MSR_PKGC3_IRTL
06_5CH. . . . . See Table 2-12
06_2AH, 06_2DH . . . . See Table 2-18
MSR_PKGC6_IRTL
06_2AH, 06_2DH . . . . See Table 2-18
MSR_PKGC7_IRTL
06_2AH. . . . . See Table 2-19
MSR_PLATFORM_BRV
0FH. . . . . See Table 2-41
MSR_PLATFORM_ENERGY_COUNTER
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_PLATFORM_ID
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-7
06_5CH . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
MSR_PLATFORM_INFO
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_3AH. . . . . See Table 2-23
06_3EH. . . . . See Table 2-24
06_3CH, 06_45H, 06_46H . . . . See Table 2-27 and Table 2-28
06_56H, 06_4FH . . . . See Table 2-34
06_57H. . . . . See Table 2-40
MSR_PLATFORM_POWER_LIMIT
06_4EH, 06_5EH. . . . . See Table 2-37
MSR_PMG_IO_CAPTURE_BASE
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_4CH. . . . . See Table 2-11
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_3AH. . . . . See Table 2-23
06_3EH. . . . . See Table 2-24
06_57H. . . . . See Table 2-40
MSR_PMH_ESCR0

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
0FH. . . . . See Table 2-41
MSR_PMH_ESCR1
0FH. . . . . See Table 2-41
MSR_PMON_GLOBAL_CONFIG
06_3EH. . . . . See Table 2-26
06_3FH. . . . . See Table 2-31
MSR_PMON_GLOBAL_CTL
06_3EH. . . . . See Table 2-26
06_3FH. . . . . See Table 2-31
MSR_PMON_GLOBAL_STATUS
06_3EH. . . . . See Table 2-26
06_3FH. . . . . See Table 2-31
MSR_POWER_CTL
06_5CH. . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
MSR_PP0_ENERGY_STATUS
06_37H, 06_4AH, 06_5AH, 06_5DH . . . . See Table 2-8
06_5CH. . . . . See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18
06_57H. . . . . See Table 2-40
MSR_PP0_POLICY
06_2AH, 06_45H . . . . See Table 2-19
MSR_PP0_POWER_LIMIT
06_4CH. . . . . See Table 2-11
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . See Table 2-18
06_57H. . . . . See Table 2-40
MSR_PP1_ENERGY_STATUS
06_5CH. . . . . See Table 2-12
06_2AH, 06_45H . . . . See Table 2-19
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
MSR_PP1_POLICY
06_2AH, 06_45H . . . . See Table 2-19
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
MSR_PP1_POWER_LIMIT
06_2AH, 06_45H . . . . See Table 2-19
06_3CH, 06_45H, 06_46H See Table 2-28
MSR_PPERF
06_4EH, 06_5EH . . . . See Table 2-37
MSR_PPIN
06_3EH. . . . . See Table 2-24
06_56H, 06_4FH . . . . See Table 2-34

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_PPIN_CTL
06_3EH.......... See Table 2-24
06_56H, 06_4FH .... See Table 2-34
MSR_R0_PMON_BOX_CTRL
06_2EH.......... See Table 2-15
MSR_R0_PMON_BOX_OVF_CTRL
06_2EH.......... See Table 2-15
MSR_R0_PMON_BOX_STATUS
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR0
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR1
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR2
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR3
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR4
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR5
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR6
06_2EH.......... See Table 2-15
MSR_R0_PMON_CTR7
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL0
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL1
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL2
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL3
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL4
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL5
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL6
06_2EH.......... See Table 2-15
MSR_R0_PMON_EVNT_SEL7
06_2EH.......... See Table 2-15
MSR_R0_PMON_IPERF0_P0

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P1
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P2
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P3
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P4
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P5
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P6
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_IPERF0_P7
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_QLX_P0
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_QLX_P1
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_QLX_P2
06_2EH. . . . . See Table 2-15
MSR_R0_PMON_QLX_P3
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_BOX_CTRL
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_BOX_OVF_CTRL
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_BOX_STATUS
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR10
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR11
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR12
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR13
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR14
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR15
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR8

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_2EH. . . . . See Table 2-15
MSR_R1_PMON_CTR9
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL10
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL11
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL12
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL13
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL14
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL15
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL8
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_EVNT_SEL9
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P10
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P11
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P12
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P13
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P14
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P15
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P8
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_IPERF1_P9
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_QLX_P4
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_QLX_P5
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_QLX_P6
06_2EH. . . . . See Table 2-15
MSR_R1_PMON_QLX_P7

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_2EH. . . . . See Table 2-15
MSR_RAPL_POWER_UNIT
06_37H, 06_4AH, 06_5AH, 06_5DH . . . . . See Table 2-8
06_4DH. . . . . See Table 2-10
06_5CH. . . . . See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H . . . . . See Table 2-18
06_3FH. . . . . See Table 2-30
06_56H, 06_4FH . . . . . See Table 2-34
06_57H. . . . . See Table 2-40
MSR_RAT_ESCR0
0FH. . . . . See Table 2-41
MSR_RAT_ESCR1
0FH. . . . . See Table 2-41
MSR_RING_PERF_LIMIT_REASONS
06_3CH, 06_45H, 06_46H . . . . . See Table 2-28
MSR_S0_PMON_BOX_CTRL
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_BOX_FILTER
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_BOX_OVF_CTRL
06_2EH. . . . . See Table 2-15
MSR_S0_PMON_BOX_STATUS
06_2EH. . . . . See Table 2-15
MSR_S0_PMON_CTR0
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_CTR1
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_CTR2
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_CTR3
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_EVNT_SEL0
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_EVNT_SEL1
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_S0_PMON_EVNT_SEL2
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_EVNT_SEL3
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S0_PMON_MASK
06_2EH. . . . . See Table 2-15
MSR_S0_PMON_MATCH
06_2EH. . . . . See Table 2-15
MSR_S1_PMON_BOX_CTRL
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_BOX_FILTER
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_BOX_OVF_CTRL
06_2EH. . . . . See Table 2-15
MSR_S1_PMON_BOX_STATUS
06_2EH. . . . . See Table 2-15
MSR_S1_PMON_CTR0
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_CTR1
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_CTR2
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_CTR3
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_EVNT_SEL0
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_EVNT_SEL1
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_EVNT_SEL2
06_2EH. . . . . See Table 2-15
06_3FH. . . . . See Table 2-31
MSR_S1_PMON_EVNT_SEL3
06_2EH. . . . . See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3FH.......... See Table 2-31
MSR_S1_PMON_MASK
06_2EH.......... See Table 2-15
MSR_S1_PMON_MATCH
06_2EH.......... See Table 2-15
MSR_S2_PMON_BOX_CTL
06_3FH.......... See Table 2-31
MSR_S2_PMON_BOX_FILTER
06_3FH.......... See Table 2-31
MSR_S2_PMON_CTR0
06_3FH.......... See Table 2-31
MSR_S2_PMON_CTR1
06_3FH.......... See Table 2-31
MSR_S2_PMON_CTR2
06_3FH.......... See Table 2-31
MSR_S2_PMON_CTR3
06_3FH.......... See Table 2-31
MSR_S2_PMON_EVNTSEL0
06_3FH.......... See Table 2-31
MSR_S2_PMON_EVNTSEL1
06_3FH.......... See Table 2-31
MSR_S2_PMON_EVNTSEL2
06_3FH.......... See Table 2-31
MSR_S2_PMON_EVNTSEL3
06_3FH.......... See Table 2-31
MSR_S3_PMON_BOX_CTL
06_3FH.......... See Table 2-31
MSR_S3_PMON_BOX_FILTER
06_3FH.......... See Table 2-31
MSR_S3_PMON_CTR0
06_3FH.......... See Table 2-31
MSR_S3_PMON_CTR1
06_3FH.......... See Table 2-31
MSR_S3_PMON_CTR2
06_3FH.......... See Table 2-31
MSR_S3_PMON_CTR3
06_3FH.......... See Table 2-31
MSR_S3_PMON_EVNTSEL0
06_3FH.......... See Table 2-31
MSR_S3_PMON_EVNTSEL1
06_3FH.......... See Table 2-31
MSR_S3_PMON_EVNTSEL2

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_3FH. . . . . See Table 2-31
MSR_S3_PMON_EVNTSEL3
06_3FH. . . . . See Table 2-31
MSR_SAAT_ESCR0
0FH. . . . . See Table 2-41
MSR_SAAT_ESCR1
0FH. . . . . See Table 2-41
MSR_SGXOWNEREPOCH0
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_SGXOWNEREPOCH1
06_5CH. . . . . See Table 2-12
06_4EH, 06_5EH . . . . See Table 2-37
MSR_SMI_COUNT
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_57H. . . . . See Table 2-40
MSR_SMM_BLOCKED
06_5CH. . . . . See Table 2-12
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
MSR_SMM_DELAYED
06_5CH. . . . . See Table 2-12
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
MSR_SMM_FEATURE_CONTROL
06_5CH. . . . . See Table 2-12
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
MSR_SMM_MCA_CAP
06_5CH. . . . . See Table 2-12
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
06_3FH. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-34
06_57H. . . . . See Table 2-40
MSR_SMRR_PHYSBASE
06_0FH, 06_17H . . . . See Table 2-3
MSR_SMRR_PHYSMASK
06_0FH, 06_17H . . . . See Table 2-3
MSR_SSU_ESCR0
0FH. . . . . See Table 2-41
MSR_TBPU_ESCR0
0FH. . . . . See Table 2-41
MSR_TBPU_ESCR1

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

0FH. . . . . See Table 2-41
MSR_TC_ESCR0
0FH. . . . . See Table 2-41
MSR_TC_ESCR1
0FH. . . . . See Table 2-41
MSR_TC_PRECISE_EVENT
0FH . . . . See Table 2-41
MSR_TEMPERATURE_TARGET
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
06_2AH, 06_2DH . . . . See Table 2-18
06_3EH. . . . . See Table 2-24
06_56H, 06_4FH. . . . . See Table 2-34
06_57H. . . . . See Table 2-40
MSR_THERM2_CTL
06_0FH, 06_17H . . . . See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H . . . . See Table 2-4
0FH. . . . . See Table 2-41
06_0EH. . . . . See Table 2-44
06_09H. . . . . See Table 2-45
MSR_THREAD_ID_INFO
06_3FH. . . . . See Table 2-30
MSR_TURBO_ACTIVATION_RATIO
06_5CH. . . . . See Table 2-12
06_3AH. . . . . See Table 2-23
06_3CH, 06_45H, 06_46H . . . . See Table 2-27
06_57H. . . . . See Table 2-40
MSR_TURBO_GROUP_CORECNT
06_5CH . . . . See Table 2-12
MSR_TURBO_POWER_CURRENT_LIMIT
06_1AH, 06_1EH, 06_1FH, 06_2EH . . . . See Table 2-13
MSR_TURBO_RATIO_LIMIT
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH . . . . See Table 2-6
06_4DH. . . . . See Table 2-10
06_5CH . . . . See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH . . . . See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH . . . . See Table 2-14
06_2EH. . . . . See Table 2-15
06_25H, 06_2CH . . . . See Table 2-16
06_2FH. . . . . See Table 2-17
06_2AH, 06_45H . . . . See Table 2-19
06_2DH. . . . . See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3EH. . . . . See Table 2-24 and Table 2-25
06_3CH, 06_45H, 06_46H . . . . See Table 2-28
06_3FH. . . . . See Table 2-30
06_3DH. . . . . See Table 2-33
06_56H, 06_4FH . . . . See Table 2-34
06_57H. . . . . See Table 2-40

MSR_TURBO_RATIO_LIMIT1
06_3EH. . . . . See Table 2-24 and Table 2-25
06_3FH. . . . . See Table 2-30
06_56H, 06_4FH . . . . See Table 2-34

MSR_TURBO_RATIO_LIMIT2
06_3FH. . . . . See Table 2-30

MSR_TURBO_RATIO_LIMIT3
06_56H. . . . . See Table 2-35
06_4FH. . . . . See Table 2-36

MSR_U_PMON_BOX_STATUS
06_3EH. . . . . See Table 2-26
06_3FH. . . . . See Table 2-31

MSR_U_PMON_CTR
06_2EH. . . . . See Table 2-15

MSR_U_PMON_CTR0
06_2DH. . . . . See Table 2-22
06_3FH. . . . . See Table 2-31

MSR_U_PMON_CTR1
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31

MSR_U_PMON_EVNT_SEL
06_2EH . . . . See Table 2-15

MSR_U_PMON_EVNTSEL0
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31

MSR_U_PMON_EVNTSEL1
06_2DH . . . . See Table 2-22
06_3FH . . . . See Table 2-31

MSR_U_PMON_GLOBAL_CTRL
06_2EH . . . . See Table 2-15

MSR_U_PMON_GLOBAL_OVF_CTRL
06_2EH . . . . See Table 2-15

MSR_U_PMON_GLOBAL_STATUS
06_2EH . . . . See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_U_PMON_UCLK_FIXED_CTL
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_U_PMON_UCLK_FIXED_CTR
06_2DH .... See Table 2-22
06_3FH .... See Table 2-31
MSR_U2L_ESCR0
0FH .... See Table 2-41
MSR_U2L_ESCR1
0FH .... See Table 2-41
MSR_UNC_ARB_PERFCTR0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_ARB_PERFCTR1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_ARB_PERFEVTSEL0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_ARB_PERFEVTSEL1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_0_PERFCTR0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_0_PERFCTR1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_0_PERFCTR2
06_2AH .... See Table 2-20
MSR_UNC_CBO_0_PERFCTR3
06_2AH .... See Table 2-20
MSR_UNC_CBO_0_PERFEVTSEL0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H See Table 2-28
06_4EH, 06_5EH .... See Table 2-38

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
MSR_UNC_CBO_0_PERFEVTSEL1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_0_PERFEVTSEL2
06_2AH .... See Table 2-20
MSR_UNC_CBO_0_PERFEVTSEL3
06_2AH .... See Table 2-20
MSR_UNC_CBO_0_UNIT_STATUS
06_2AH .... See Table 2-20
MSR_UNC_CBO_1_PERFCTR0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_1_PERFCTR1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_1_PERFCTR2
06_2AH .... See Table 2-20
MSR_UNC_CBO_1_PERFCTR3
06_2AH .... See Table 2-20
MSR_UNC_CBO_1_PERFEVTSEL0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_1_PERFEVTSEL1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_1_PERFEVTSEL2
06_2AH .... See Table 2-20
MSR_UNC_CBO_1_PERFEVTSEL3
06_2AH .... See Table 2-20
MSR_UNC_CBO_1_UNIT_STATUS
06_2AH .... See Table 2-20
MSR_UNC_CBO_2_PERFCTR0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_2_PERFCTR1
06_2AH .... See Table 2-20

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_2_PERFCTR2
06_2AH .... See Table 2-20
MSR_UNC_CBO_2_PERFCTR3
06_2AH .... See Table 2-20
MSR_UNC_CBO_2_PERFEVTSEL0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_2_PERFEVTSEL1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_2_PERFEVTSEL2
06_2AH .... See Table 2-20
MSR_UNC_CBO_2_PERFEVTSEL3
06_2AH .... See Table 2-20
MSR_UNC_CBO_2_UNIT_STATUS
06_2AH .... See Table 2-20
MSR_UNC_CBO_3_PERFCTR0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_3_PERFCTR1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_3_PERFCTR2
06_2AH .... See Table 2-20
MSR_UNC_CBO_3_PERFCTR3
06_2AH .... See Table 2-20
MSR_UNC_CBO_3_PERFEVTSEL0
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_3_PERFEVTSEL1
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_CBO_3_PERFEVTSEL2
06_2AH .... See Table 2-20

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_UNC_CBO_3_PERFEVTSEL3
06_2AH .... See Table 2-20
MSR_UNC_CBO_3_UNIT_STATUS
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFCTR0
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFCTR1
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFCTR2
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFCTR3
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFEVTSEL0
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFEVTSEL1
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFEVTSEL2
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_PERFEVTSEL3
06_2AH .... See Table 2-20
MSR_UNC_CBO_4_UNIT_STATUS
06_2AH .... See Table 2-20
MSR_UNC_CBO_CONFIG
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_PERF_FIXED_CTR
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_PERF_FIXED_CTRL
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_PERF_GLOBAL_CTRL
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38
MSR_UNC_PERF_GLOBAL_STATUS
06_2AH .... See Table 2-20
06_3CH, 06_45H, 06_46H .... See Table 2-28
06_4EH, 06_5EH .... See Table 2-38

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

MSR_UNCORE_ADDR_OPCODE_MATCH
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_FIXED_CTR_CTRL
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_FIXED_CTR0
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERF_GLOBAL_CTRL
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERF_GLOBAL_STATUS
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL0
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL1
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL2
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL3
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL4
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL5
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL6
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PERFEVTSEL7
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC0
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC1
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC2
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC3
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC4
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
MSR_UNCORE_PMC5
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .... See Table 2-14
06_2EH .... See Table 2-15
MSR_UNCORE_PMC6

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH . . . . . . See Table 2-14
MSR_UNCORE_PMC7
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH . . . . . . See Table 2-14
MSR_UNCORE_PRMRR_BASE
06_4EH, 06_5EH . . . . . . See Table 2-37
MSR_UNCORE_PRMRR_MASK
06_4EH, 06_5EH . . . . . . See Table 2-37
MSR_W_PMON_BOX_CTRL
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_BOX_OVF_CTRL
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_BOX_STATUS
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_CTR0
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_CTR1
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_CTR2
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_CTR3
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_EVNT_SEL0
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_EVNT_SEL1
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_EVNT_SEL2
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_EVNT_SEL3
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_FIXED_CTR
06_2EH . . . . . . See Table 2-15
MSR_W_PMON_FIXED_CTR_CTL
06_2EH . . . . . . See Table 2-15
MSR_WEIGHTED_CORE_C0
06_4EH, 06_5EH . . . . . . See Table 2-37
MTRRfix16K_80000
06_0EH . . . . . . See Table 2-44
P6 Family . . . . . . See Table 2-46
MTRRfix16K_A0000
06_0EH . . . . . . See Table 2-44
P6 Family . . . . . . See Table 2-46
MTRRfix4K_C0000

MSR Name and CPUID DisplayFamily_DisplayModel **Location**
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_C8000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_D0000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_D8000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_E0000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_E8000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_F0000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix4K_F8000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRfix64K_00000
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase0
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase1
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase2
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase3
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase4
06_0EH ... See Table 2-44
P6 Family ... See Table 2-46
MTRRphysBase5

MSR Name and CPUID DisplayFamily_DisplayModel **Location**

06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysBase6
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysBase7
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask0
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask1
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask2
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask3
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask4
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask5
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask6
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
MTRRphysMask7
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46
ROB_CR_BKUPTMPDR6
06_0EH . . . . See Table 2-44
P6 Family . . . . See Table 2-46